3. Context Free Grammar

Formal Languages

Grammars

Classification of Grammar

Chomsky Hierarchy Theorem

Context Free Grammar

Leftmost & RightMost Derivation

Parse Tree

Ambiguous Grammar

Simplification of Context Free Grammar - elimination of Useless Symbol

E-production and Unit Productions

Normal forms of ContextFree Grammar - Chomsky Normal Form & Greibach Normal Form.

Pumping Lemma

Closure properties

Application of Context Free Grammar

Definition of Context Free Grammar:

In Formal Language, CFG generates Context Free Language (or) Set of CFL are accepted by Push Down Automata

CFG $\xrightarrow{\text{Generate}}$ CFL $\xrightarrow{\text{Accepted}}$ PDA

Context Free Grammar is defined by 4 tuples

$G = (N, T, P, S)$ $\quad N \longleftrightarrow V$, $T \longleftrightarrow \Sigma$

where
- N = Set of Non-Terminals
- T = Set of Terminals
- P = Production rule
- S = Start Symbol

The production rule in CFG is in form of

$$\alpha \longrightarrow \beta$$

where $\alpha \in NT$ (non terminal)

$|\alpha| = 1$

$\beta \in (NT \cup T)^*$

Who discovered all those production rules used here?

Backus - Naur Form:

In 1960, John Backus and Peter Naur introduce Formal notation method for describing Syntax of programming language, which is known as Bacus Naur Form (or) Simply BNF.

Basically, this BNF is designed for ALGOL-60

BNF and context free grammar were nearly identical.

BNF is metalanguage for programming language.

Metalanguage is a language that is used to describe another language

BNF symbols to describe a syntax

- ::= "is defined as"
- < > means "can be defined as" / described
- | means "or"

Rules in BNF:

LHS → RHS

<NT> → {T} {<NT>}

Start symbol

(or)

<NT> ::= {T} {<NT>}

Problem:

Construct a CFG for the langue $L = \{ wcw^R \mid w \in (a,b)^* \}$

(4) $L = \{ wcw^R \mid w \in (a,b)^* \}$

here w = string of a's & b's

w^R = reverse of w string.

$L = \{ c, aca, bcb, abcba, bacab, \ldots \}$

CFG defines 4 tuples

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

$N = \{S\}$

$T = \{a, b, c\}$

Production rule for language:

$S \longrightarrow c$

$S \longrightarrow aSa$

$S \longrightarrow bSb$

$S = \{S\}$

Derivations and Parse tree:

Derivation

28/12/21 (clan)

Left recursion:- ↔ direct left recursion / indirect left recursion

$A \xrightarrow{CFG} A\alpha$ → ex of direct left recursion

If LHS & RHS leftmost is same then we get left recursion

left recursion is not supported by grammar. So we can remove this by adding two new production rules.

Eg: $A \longrightarrow A\alpha / \beta$

$A \longrightarrow \beta A'$

$A' \longrightarrow \alpha A' / \epsilon$ } equivalent to.

Indirect left recursion:

$A \longrightarrow B\alpha / \beta$

$B \longrightarrow A\beta / \delta$

$A \longrightarrow A\beta\alpha / \beta / \delta\alpha$

$A \longrightarrow \beta A'$

$A' \longrightarrow \beta\alpha A' / \epsilon$

$A \longrightarrow \delta\alpha A'$

Left factoring — Overhead · Backtracking

$A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \ldots \alpha\beta_n$

$A \to \alpha A'$

$A' \to \beta_1 | \beta_2 | \beta_3 | \ldots \beta_n$

① $A \to abd | abe | abf | cfgx | cfgy | cfgz$

$A \to abA'$, $A \to cfgA''$

$A' \to d | ef$ $\quad A'' \to x | y | z$

Simplification of Context Free Grammar

1) Reduce the complexity of program by reducing production rules.

2) Easily generate grammar with less no. of production rules.

how we can simply:-

1) by eliminating useless Symbols → Non-generating Symbols, Non-reachable Symbols

2) by eliminating unit production

3) by eliminating ϵ-production.

A non-terminal which doesn't produce a terminal = Non-generating Symbol.

If it is available in start symbol, we can't remove it if we have only one start symbol.

$S \to AB$

$B \to b$

$S \to AB$

$A \to aB$

$B \to b$

$C \to AB$ — here C is Non-reachable Symbol.

Unit production:

$A \to D$, $D \to C$, $C \to a$ → $A \to a$

$B \to C$, $C \to b$ → $B \to b$

$S \to AB$

DB

EB

aB

ac

ab

↓ without unit production

$S \to AB$

$S \to aB$

$S \to ab$

↓ with unit production

ϵ-production:-

① $S \to Ab$

$A \to BA$

$A \to a$

$A \to \epsilon$

→ $S \to b | ab$

$A \to B | Ba$

$A \to a$

② $S \to aA$

$A \to b | \epsilon$

↓

$S \to ab | a$

③ $S \to AB$

$A \to \epsilon$

$B \to C$

$C \to D$

$D \to b$

$E \to a$

$S \to ab$

$S \to AC$
$S \to BA$
$C \to CB$
$C \to AC$
$A \to a$
$B \to aC|b$

$S \to aC$
$\to aCB$
$\to aACB$
$\to aaCb$

$S \to BA$
$A \to a$
$B \to b$

after removing non-generating

Normal forms:

$G_1 \to L \to m/c$ (Pushdown Automata)

complex

CFG → Parse tree

CNF = Chomsky Normal Form

GNF = Greibach Normal Form

CFG → CNF → GNF → M/C = easy process

CNF / GNF

We get binary Parse tree

CNF = NT → exactly two NT
NT → Single terminals

i.e
$S \to AB$
$A \to CC$
$B \to BA$
$A \to a$
$B \to b$
$C \to c$

CNF:

Step1 : Eliminate useless units

Step2 :- If
NT → two NT
NT → Single T
Stop
else go to next step

Step3 :- $NT \to T_1 T_2 T_3 \dots T_n$ → convert to step4 ($\because A \to abc$)

Step4 :- $NT \to NT_1 NT_2 NT_3 \dots$ → $NT \to NT_1 A_1$, $A_1 \to NT_1 A_2$, $A_2 \to NT_3 A_3$

Step5 :- $NT \to T_1 T_2 \dots T_n, NT_1 NT_2 \dots NT_n$

$A \to abc$
$C_a \to a$
$C_b \to b$
$C_c \to c$
$A \to C_a C_b C_c$
$A \to A' C_a$
$A' \to C_b C_c$

$S \to ABa$
$A \to aab$
$B \to Ac$

Consider
$C_a \to a$
$C_b \to b$
$C_c \to c$

$S \to ABC_a$
$A \to C_a C_a C_b$
$B \to AC_c$

$S \to AD$
$D \to BC_a$
$A \to C_a C_a C_b$
$A \to C_a E$
$E \to C_a C_b$

$S \to AD$
$A \to C_a E$
$B \to AC_c$
$D \to BC_a$
$E \to C_a C_b$

GNF

NT → Single terminal & string of TUNT

NT → Single terminal

$A_i \rightarrow A_j x$

$i < j$ No problem

$i \geq j$ Problem

$S \rightarrow AA \mid a$
$A \rightarrow SS \mid b$ } i is in CNF

$S \rightarrow A_1$

$A \rightarrow A_2$

$A_1 \rightarrow A_2 A_2 \mid a$ $i < j$ ✓

$A_2 \rightarrow A_1 A_1 \mid b$ hu $i > j$ ✗

$A_2 \rightarrow A_2 A_2 A_1 \mid aA_1 \mid b$ now $i = i$ $i < z$

Now introduce new terminal Z which is remaining terms other than equal NT.

$Z \rightarrow A_2 A_1 \mid A_2 A_1 Z$

$A_2 \rightarrow aA_1 Z \mid bZ \mid aA_1 \mid b$

$A_1 \rightarrow aA_1 Z A_2 \mid bZ A_2 \mid aA_1 A_2 \mid bA_2 \mid a$

$Z \rightarrow aA_1 Z A_1 \mid bZ A_1 \mid aA_1 A_1 \mid bA_1 \mid a_1 A_1 Z A_1 Z \mid bZ A_1 Z \mid aA_1 A_1 Z \mid bA_1 Z$

CFG is not closed under Complementary.

Applications:-

CFG 1) It is used in Syntax Analyzer

Ex:- $S \rightarrow a \mid aA \mid B$

$A \rightarrow aBB \mid \epsilon$

$B \rightarrow Aa \mid b \mid a$

after Remove ϵ-production

$S \rightarrow a$.
$S \rightarrow aA$ ✗
$S \rightarrow B$ ✗
$A \rightarrow aBB$ ✗
$B \rightarrow Aa$ ✗
$B \rightarrow b$
$B \rightarrow a$

~~Let~~ ~~$C_a = a$~~

$S \rightarrow a$
$S \rightarrow C_a A$
$S \rightarrow B$ ✗
$A \rightarrow aD$
$D \rightarrow BB$
$B \rightarrow ACa$
$B \rightarrow b$
$B \rightarrow a$

$S \rightarrow a$
$S \rightarrow C_a A$
$S \rightarrow b \mid a$
$A \rightarrow aD$
$D \rightarrow BB$
$B \rightarrow ACa$
$B \rightarrow a \mid b$
$C_a \rightarrow a$

$S \rightarrow a \mid C_a A \mid b$
$A \rightarrow aD$
$D \rightarrow BB$
$B \rightarrow AC_a \mid a \mid b$
$C_a \rightarrow a$

GNF :- $A_1 \longrightarrow A_2 A_3$ ✓

$A_2 \longrightarrow A_3 A_1 \mid b$ ✓

$A_3 \longrightarrow A_1 A_2 \mid a$ ✗

Substitute A_1 value

$A_3 \longrightarrow A_2 A_3 A_2 \mid a$

Substt A_2

$A_3 \longrightarrow A_3 A_1 A_3 A_2 \mid b A_3 A_2 \mid a$

$Z \rightarrow A_1 A_3 A_2 \mid A_1 A_3 A_2 Z$

$A_3 \longrightarrow$ Substitut $bA_3A_2 \mid a$ in $A_3A_1A_3A_2$

$A_3 \rightarrow b A_3 A_2 A_1 A_3 A_2 \mid a A_1 A_3 A_2$
($A_1A_3A_2$ = Z)

$\rightarrow b A_3 A_2 Z \mid aZ$

$A_3 \longrightarrow bA_3A_2Z \mid aZ \mid bA_3A_2 \mid a$

$A_2 \longrightarrow bA_3A_2ZA_1 \mid aZA_1 \mid bA_3A_2A_1 \mid aA_1 \mid b.$

$A_1 \rightarrow bA_3A_2ZA_1A_3 \mid aZA_1A_3 \mid bA_3A_2A_1A_3 \mid aA_1A_3 \mid bA_3.$

$Z \rightarrow$

21/12/21

Unit-III

Definition of Context Free Grammar:

According to the Chomsky Hierarchy, context free grammar or in short CFG is type2 Grammar.

An Mathematical description, we can describe it as all production are in the form $\alpha \longrightarrow \beta$

* where $\alpha \in V_N$ (Set of non-terminals)
  $|\alpha| = 1$ i.e there will be only one Nonterminal at the left hand side (LHS)
* $\beta \in V_N \cup \Sigma$ (Set of non-terminals & terminals)

{A string consists of atleast one non-terminal} $\longrightarrow$ {A string of terminals and non-terminals}

Backus Naur Form (BNF)

This is a formal notation to describe the syntax of a given language

This notation was first introduced by John Backers and Peta Naur.

The Symbols used in BNF are as follows:

::= denotes "is defined as"

| denotes "Or"

< > used to hold category names

Eg: <number> ::= <digit> | <number> <digit>

<digit> ::= 0|1|2|3|4|5|6|7|8|9

This can be described as no is a digit or a num followed by a digit.

A digit is any of the characters from 0 to 9.

From BNF comes the extended BNF which uses some notations of regular expression such as + or *

As an example, the identifier of a programming language is defined as

<identifier> ::= letter (letter | digit)*

<letter> ::= a|...|z|A|...|Z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

BNF is widely used in CFG.

1) Construct a CFG for the language $L = \{ wcw^R \mid w \in (a+b)^* \}$

w is a string of any combination of a & b.

So w^R is also a string of any combination of a and b, but it is a string which is reverse of w. c is a terminal symbol like a, b.

$L = \{ c, aca, bcb, abcba, bacab, abbcbba, \ldots \}$

$w \in (a,b)^*$

The null symbols are also accepted in the place of a, b. That means only c is accepted by thes language set.

∴ The production rules are

$$S \rightarrow aSa \mid bSb \mid c$$

The grammar becomes $G_L = \{ V_N, T, P, S \}$.

where $V_N = \{S\}$

$T / \Sigma = \{a, b, c\}$

$P = S \rightarrow aSa \mid bSb \mid c$

$S = \{S\}$

2) Construct a CFG for the RE $(0+1)^* 0 1^*$

Any combination of 0 and 1 followed by a single 0 and ending with any number of 1.

In RE, a single zero is between $(0+1)^*$ and 1^*. Consider two NT A & B which can be replaced multiple times.

The production rules P of the grammar for constructing this regular expression are:

$$S \rightarrow ASB \mid 0$$

$$A \rightarrow 0A \mid 1A \mid \epsilon$$

$$B \rightarrow 1B \mid \epsilon$$

The grammar is $G_L = (V_N, T, P, S)$

where, $V_N = \{S, A, B\}$

$T = \{0, 1\}$

$P = \{ S \rightarrow ASB \mid 0,\ A \rightarrow 0A \mid 1A \mid \epsilon,\ B \rightarrow 1B \mid \epsilon \}$

$S = \{S\}$

3) Construct a CFG for the regular expression $(011+1)^*(01)^*$

The regular expression consists of two parts $(011+1)^*$ $(01)^*$.

Here from the start symbol, two NT each of them producing one part, are taken $(011+1)$ can be written as $011 \mid 1$

As * represents any combination, null string is also included in the language set

The production rules (P) of the grammar are:

$$S \longrightarrow BC$$
$$B \longrightarrow AB \mid \varepsilon$$
$$A \longrightarrow 011 \mid 1$$
$$C \longrightarrow DC \mid \varepsilon$$
$$D \longrightarrow 01$$

The grammar is $G = \{V_N, \Sigma, P, S\}$

where $V_N = \{S, A, B, C, D\}$

$\Sigma = \{0, 1\}$

$S = \{S\}$

4) Construct a CFG for a string in which there are equal number of binary numbers.

$$S \longrightarrow 0S1 \mid 1S0 \mid \varepsilon$$

The grammar is $G = \{V_N, \Sigma, P, S\}$

$V_N = \{S\}$

$\Sigma = \{0, 1\}$

$S = \{S\}$

Derivation and Parse Tree:

derivation:

In the process of generating a language from the given production rules of grammar, the NT are replaced by corresponding strings of the right hand side (RHS) of the production

But if there are more than one NT, then which of the ones will be replaced must be determined.

Depending on the selection, the derivation is divided into 2 parts

1) LMD (left Most derivation)
2) RMD (Right Most derivation)

1) Left Most Derivation:

A derivation is called Left Most Derivation if we replace only the left most non-terminal by some production rules at each step of the generating process of the language from the grammar.

2) Right Most Derivation:

A derivation is called Right Most Derivation if we replace only the right most NT by some production rules at each step of the generating process of the language from the grammar

Eg:-

1) Construct the string 0100110 from the following grammar by using 1) LMD 2) RMD

$S \rightarrow 0S \mid 1AA$

$A \rightarrow 0 \mid 1A \mid 0B$

$B \rightarrow 1 \mid 0BB$

Ans i) Left Most Derivation

$S \rightarrow 0S$

$\rightarrow 01\underline{A}A$

$\rightarrow 010\underline{B}A$

$\rightarrow 0100\underline{B}BA$

$\rightarrow 01001\underline{B}A$

$\rightarrow 0100111\underline{A}$

$\rightarrow 0100110$

NT that are replaced are underlined.

ii) Right Most Derivation

$S \rightarrow 0S$

$\rightarrow 01A\underline{A}$

$\rightarrow 01\underline{A}0$

$\rightarrow 010\underline{B}0$

$\rightarrow 0100\underline{B}B0$

$\rightarrow 0100B10$

$\rightarrow 0100110$

3) Construct the string abbbb from the grammar given below by using i) LMD ii) RMD

$S \rightarrow aAB$
$A \rightarrow bBb$
$B \rightarrow A/\varepsilon$

i) LMD:-

$S \rightarrow a\underline{A}B$
$\rightarrow abBbB$
$\rightarrow ab\underline{A}bB$
$\rightarrow abbBbbB$
$\rightarrow abbbb$

ii) RMD:-

$S \rightarrow aA\underline{B}$
$\rightarrow a\underline{A}A$
$\rightarrow aAbBb$
$\rightarrow aAb\varepsilon b$
$\rightarrow aAbb$
$\rightarrow abBbbb$
$\rightarrow ab\varepsilon bbb$
$\rightarrow abbbb$

Parse Tree :-

Parsing a string is finding a derivation for that string from a given grammar.

A parse tree is a tree representation of deriving a CFL from a given context grammar. These type of trees are called as derivation trees.

* A parse tree is an ordered tree in which the LHS of the production represents a parent node and the RHS of a production represents a children node.
* There are certain conditions for constructing a parse tree from a given CFG. Those are as follows:
  - → Each vertex of the tree must have a label. The label is a non-terminal or terminal or null
  - → The root of the tree is the start symbol 'S'.
  - → The label of internal vertices is a NT $\in V_n$
  - → If there is a production $A \rightarrow X_1 X_2 X_3 \ldots X_k$, then for a vertex label A, the children of the node will be $X_1 X_2 \ldots X_k$
  - → A vertex n is called a leaf of parse tree if its label is a terminal symbol $\in \Sigma$ or null (Λ)

* We are describing the similar properties of a tree and a CFG
* For a tree, there must be some root. For every CFG, there is a single start symbol
* Each node of a tree has a single label. For every CFG at LHS there is a single NT
* A child node is derived from a single parent. For constructing a CFL, a NT is replaced by a suitable string at RHS. Each of the character of the string is generating a node. That is, for each single node there is a single parent.
* For these similarities, a parse tree can be generated only for a CFG

Note :- A parse tree is not possible with context sensitive Grammar because there are productions such as $bB \rightarrow aa$ (or) $AA \rightarrow B$

1) Find the parse tree for generating the string 0100110 from the following grammar.

$S \rightarrow 0S \mid 1AA$

$A \rightarrow 0 \mid 1A \mid 0B$

$B \rightarrow 1 \mid 0BB$

A) For generating the string 0100110 from the given CFG, the leftmost derivations will be

$S \rightarrow 0\underline{S}$

$\rightarrow 01\underline{A}A$

$\rightarrow 010\underline{B}A$

$\rightarrow 0100\underline{B}BA$

$\rightarrow 01001\underline{B}A$

$\rightarrow 0100110$

Parse tree:

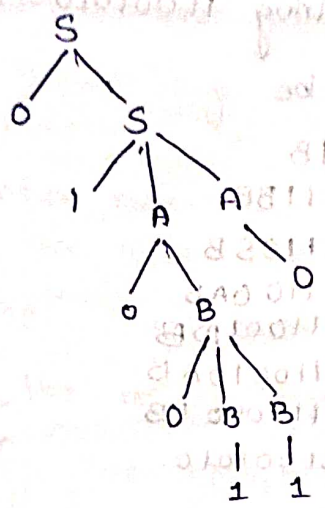


fig: parse tree for LMD

For generating the string 0100110 from the given CFG the rightmost Derivation will be.

$S \rightarrow 0\underline{S}$

$\rightarrow 01A\underline{A}$

$\rightarrow 01\underline{A}0$

$\rightarrow 010\underline{B}0$

$\rightarrow 0100B\underline{B}0$

$\rightarrow 0100\underline{B}10$

$\rightarrow 0100110$

Parse tree:

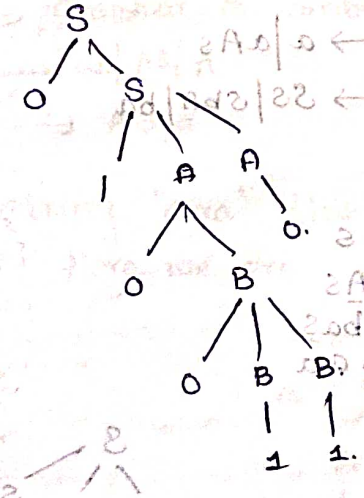


fig: parse tree for RMD.

) Find the parse tree for generating the string 11001010 from the given grammar

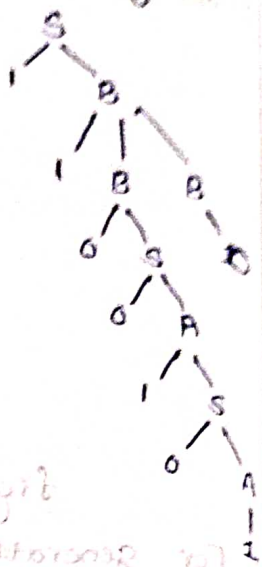
$S \rightarrow 1B \mid 0A$

$A \rightarrow 1 \mid 1S \mid 0AA$

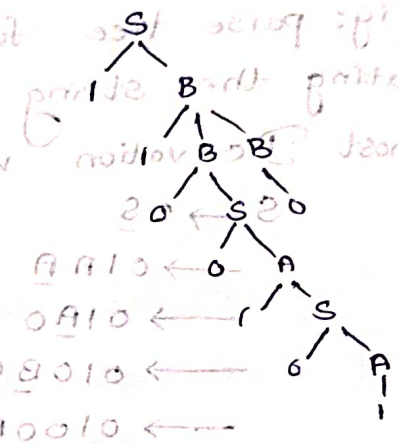
$B \rightarrow 0 \mid 0S \mid 1BB$

2) For generating the string 11001010 from the given CFG, the LMD will be

$S \longrightarrow 1B$
$\longrightarrow 11BB$
$\longrightarrow 110SB$
$\longrightarrow 1100AB$
$\longrightarrow 11001SBB$
$\longrightarrow 110010AB$
$\longrightarrow 1100101B$
$\longrightarrow 11001010.$



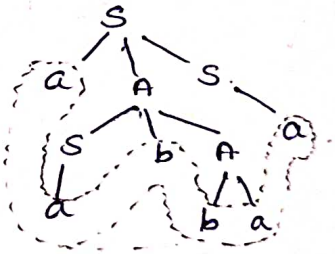
RMD: $S \longrightarrow 1B$
$\longrightarrow 11BB$
$\longrightarrow 11B0$
$\longrightarrow 110S0$
$\longrightarrow 1100A0$
$\longrightarrow 11001S0$
$\longrightarrow 110010A0$
$\longrightarrow 11001010$



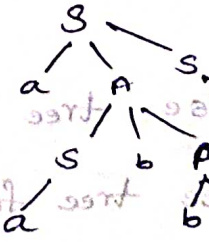
3) Construct the parse tree for the string aabbaa from the following grammar.

$S \longrightarrow a \mid aAs$
$A \longrightarrow SS \mid SbA \mid ba$

(A) LMD:- $S \longrightarrow aAs$
$\longrightarrow asbAs$
$\longrightarrow aabAS$
$\longrightarrow aabbaS$
$\longrightarrow aabbaa$



RMD:- $S \longrightarrow aAS$
$\longrightarrow aAa$
$\longrightarrow asbAa$
$\longrightarrow asbbaa$
$\longrightarrow aabbaa$



Ambiguity in Context free grammars:

For generating a string from a given grammar, we have to derive the string step by step from the production rule of the given grammar. For this derivation, we know two types of derivations: i) LMD & ii) RMD. Exept these two there is also another approach called the mixed approach

→ Here particularly the LMD/RMD is not maintained in each step whereas any of the NT present in deriving string is replaced by suitable production rule

By this process, different types of derivation can be generated from deriving may be the same/different.

⇒ A grammar of a language is called ambiguous if any of the cases for generating a particular string more than one parse tree can be generated.

⇒ The following examples describe the ambiguity in CFG:

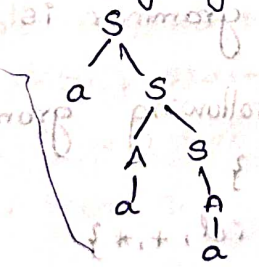
1) check whether the grammar is ambiguous or not

$S \longrightarrow aS \mid AS \mid A$

$A \longrightarrow AS \mid a$

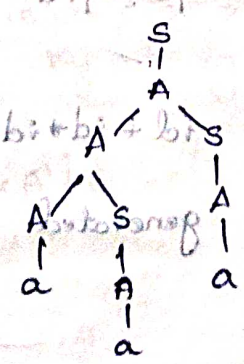
Consider the string "aaa" the string can be generated in many ways. Here we are going two ways

i) $S \rightarrow AS$
$\rightarrow aAS$
$\rightarrow aaA$
$\rightarrow aaa$



ii) RMD:-

$S \rightarrow AS$
$\rightarrow ASS$
$\rightarrow aSS$
$\rightarrow aAS$
$\rightarrow aaS$
$\rightarrow aaA$
$\rightarrow aaa$



Here for the same string derived from the same grammar we are getting more than one parse tree. So, according to definition, the grammar is an ambiguous grammar.

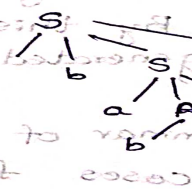
2) Prove that the following grammar is ambiguous.

$S \longrightarrow a \mid abSb \mid aAb$

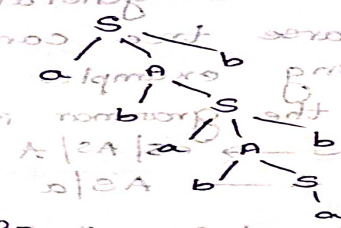
$A \longrightarrow bS \mid aAAb$

A) Take a string ababbabb. The string can be generated in the following ways

LMD:- $S \longrightarrow abSb$
$\longrightarrow ababAbb$
$\longrightarrow ababSbb$
$\longrightarrow ababbabb$



RMD:- $S \longrightarrow aAb$
$\longrightarrow abSb$
$\longrightarrow abaAbb$
$\longrightarrow ababSbb$
$\longrightarrow ababbabb$



As we are getting two-parse trees of generating a string from the given grammar, the grammar is ambiguous

3) Prove that the following grammar is ambiguous

$V_N = \{S\}$

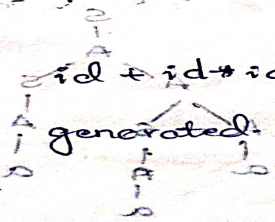
$\Sigma = \{id, +, *\}$

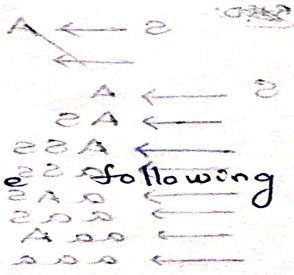
$P: S \longrightarrow S+S \mid S*S \mid id$

$S = \{S\}$

A) let us take a string id+id*id

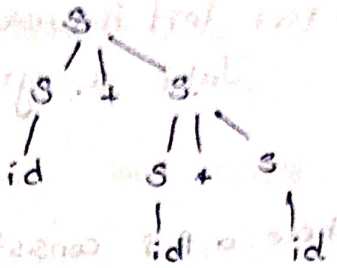
The string can be generated in the following ways.





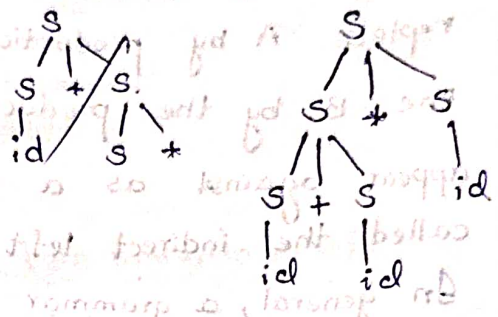
LMD

$S \rightarrow \underline{S}+S$
$\rightarrow id + S$
$\rightarrow id + \underline{S}+S$
$\rightarrow id + id + S$
$\rightarrow id + id + id$.



RMD

$S \rightarrow \underline{S}+S$
$\rightarrow \underline{S}+S+S$
$\rightarrow id + S + S$
$\rightarrow id + id + S$
$\rightarrow id + id + id$.



As we are getting two parse trees for generating a string from the given grammar, the grammar is ambiguous

Left Recursion & Left factoring:

Left Recursion:

A context free grammar is called left recursive if a non-terminal 'A' as a leftmost symbol appears or alternatively at the time of derivation either immediately

→ In other words, a grammar is left recursive if it has a NT 'A' such that there is a derivation.

$A \overset{+}{\Rightarrow} A\alpha$ for something α

Direct left Recursion:

Let the grammar be $A \rightarrow A\alpha \mid \beta$, where α & β consists of terminal and but β doesn't starts with A

At time of derivation, if we start from A and replace A by production $A \rightarrow A\alpha$. then for all time A will be the leftmost NT Symbol

A left recursion is not supported by grammar. So we can remove this by adding new production rules

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \epsilon$

Indirect left Recursion:

Take a grammar in the form

$$A \longrightarrow B\alpha \mid \beta$$
$$B \longrightarrow AB \mid \delta$$

where α, B, β consists of terminals | NT.

At the time of derivation if we start from A, replace A by production $A \longrightarrow B\alpha$ & after that replace the B by the production $B \longrightarrow AB$ then A A will appear against as a leftMost NT symbol. This is called the indirect left recursion

In general, a grammar in the form

$$A_0 \longrightarrow A_1\alpha_1 \mid \ldots$$
$$A_1 \longrightarrow A_2\alpha_2 \mid \ldots$$
$$A_n \longrightarrow A_0\alpha_{(a<i)} \mid \ldots$$

is the left recursive grammar

Note:

The left recursive grammar is to be converted into a non-left recursive grammar because the top down parsing technique can't handle left recursion.

Immediate left recursion can be removed by the following process. This is called the Moore's proposal.

For a grammar in the form

$A \longrightarrow A\alpha \mid \beta$ where β doesn't start with A

the equivalent grammar after removing left recursion is

$$A \longrightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$$
$$A' \longrightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

For indirect left recursion removal, there is algorithm

Arrange non-terminals in some order: $A_1 \ldots A_n$

```
-for i from 1 to n do {
    -for j from 1 to i-1 do {
        replace each production
            Ai ⟶ Aj γ by
            Ai ⟶ α1 γ | .. | αn γ
        where
            Aj ⟶ α1 | α2 .. | αk
    }
```

- eliminate immediate left recursion among A_i productions

}

Consider the following example to make the moore's proposal Clear

Remove the left recursion from the following grammar

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id(E)$

In the grammar there are two immediate left recursion $E \rightarrow E+T$ and $T \rightarrow T*F$

By using Moore's proposal the left recursion $E \rightarrow E+T$ is removed as

$E \rightarrow TE'$ and $E' \rightarrow +TE' \mid \varepsilon$

the left recursion $T \rightarrow T*F$ is removed as

$T \rightarrow FT'$

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

The CFG after removing the left recursion becomes

$E \rightarrow TE'$

$E' \rightarrow TE' \mid \varepsilon$

$T \rightarrow FT'$

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

$F \rightarrow id(E)$

Left Factoring:

Let us assume that is in a grammar there is a production rule in the form $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$ The parses generated from this kind of grammar is not efficient as it requires backtracking. To avoid this problem we need left factor for grammar.

After left factoring, the previous grammar is transformed into

$A \rightarrow \alpha A_1$

$A_1 \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \ldots \mid \beta_n$

1) Left factor the following grammar

$A \rightarrow abB | aB | cdg | cdcB | cdfB$

A) In the previous grammar, the RHS productions abB and aB both start with 'a'. So they can be left factorized. In the same way, cdg, cdcB, cdfB all start with 'cd'. So they can be left factorized.

$A \rightarrow aA'$

$A' \rightarrow bB | B$

$A \rightarrow cdA''$

$A'' \rightarrow g | cB | fB$

Removing Ambiguity:-

→ An Ambiguous grammar, may be converted into an unambiguous grammar by implementing the precedence and associativity constraints.

These constraints are implemented using the following rules:-

Rule-1:-

The precedence of the operators is implemented using the following rules:

→ The level at which the production is present defines the priority of the operator contained in it

→ The higher the level of the production, the lower the priority of operator

→ The lower the level of production, the higher the priority of operator

Rule-2:-

The associativity of operators is implemented by using the following rules:-

→ The operator is left associative, induce left recursion in it's production

→ If the operator is right associative, induce right recursion in its production

Simplification of Context-Free Grammar:

CFG may contain different types of useless symbols, unit productions and null production. These types of symbols and productions increase the number of steps in generating a language from a CFG. Reduced Grammar contains less number of non-terminals and productions. So the time complexity for the language generated process between less from the reduced grammar

CFG can be simplified in the following process:

1. Removal of useless symbols
   a) Removal of non-generating Symbols.
   b) Removal of non-reachable Symbols.
2. Removing unit production Symbols
3. Removing Null production

Removal of Useless Symbols:

Useless Symbols are of two types:

1) Non generating Symbol: They are those symbols in which do not produce any terminal string.
2) Non-reachable Symbol: Those symbols which cannot be reached at any time starting from the Start Symbol.

These two types of useless symbols can be removed according to the following process:

→ Find the non-generating symbols i.e the symbols which do not generate any terminal string. If the start symbol is found non-generating leave that symbol. Because the start symbol cannot be removed, the language generating process starts from Symbol

→ For removing non-generating symbol, remove those production whose right side and left side contain those Symbols

→ Now find the non-reachable symbols i.e the symbols which cannot be reached, starting from the start symbol

→ Remove non-reachable symbols such as, rule(2)

1) Remove the useless symbols from the given CFG

$S \rightarrow A$

$S \rightarrow BA$

$C \rightarrow CB$

$C \rightarrow AC$

$A \rightarrow a$

$B \rightarrow aC|b$

A) There are two types of useless symbols: non-generating & non-reachable symbols. First, we are finding non-generating symbols.

→ Those symbols which do not produce any terminal string are non-generating symbol as it does not produce any terminal string

→ So, we have to remove the symbol C. To remove C, all the productions containing C as a symbol must be removed. By removing the productions, the minimized grammar will be

$S \rightarrow BA$

$A \rightarrow a$

$B \rightarrow b$

→ Now we have to find non-reachable symbols, the symbols which cannot be reached at any time starting from start symbol

→ There is no non-reachable symbol in the grammar. So the minimized form of the grammar by removing useless symbols is

$S \rightarrow BA|A$

$A \rightarrow a$

$B \rightarrow b$

Removal of Unit Productions:

Production in the form Non-terminal → Single non-terminal is called unit production.

Unit production increases the number of steps as well as the complexity at the time of generating language from the grammar. This will be clear if we take an example

Let there be a grammar

$S \rightarrow AB$, $A \rightarrow E$, $B \rightarrow C$, $C \rightarrow D$, $D \rightarrow b$, $E \rightarrow a$

From here, if we are going to generate a language, then it will be generated by the following way:

$$S \rightarrow AB \rightarrow \in B \rightarrow aB \rightarrow aC \rightarrow aD \rightarrow ab.$$

The grammar by removing unit production and as well as minimizing will be

$$S \rightarrow AB$$
$$A \rightarrow a$$
$$B \rightarrow b$$

From this the language will be generated by the following way:

$$S \rightarrow AB \rightarrow aB \rightarrow ab$$

1) Remove the unit productions from the following grammar.

$$S \rightarrow AB$$
$$A \rightarrow E$$
$$B \rightarrow C$$
$$C \rightarrow D$$
$$D \rightarrow b$$
$$E \rightarrow a$$

A) In the given grammar, the unit productions are:

$$A \rightarrow E$$
$$B \rightarrow C$$
$$C \rightarrow D.$$

For removing the unit production $A \rightarrow E$, we have to find a non-unit production in the form

$E \rightarrow$ Some string of terminal or NT or both.

There is a production. $E \rightarrow a$

So, $A \rightarrow a$ will be added to the production rule.

Similarly $C \rightarrow b$ will be added to the production rule.

Now, the modified production rules are:

$$S \rightarrow AB$$
$$A \rightarrow a$$
$$B \rightarrow b$$
$$C \rightarrow b$$
$$D \rightarrow b$$
$$E \rightarrow a$$

This is the grammar without unit production but it is not minimized.

There are useless symbols (non-reachable symbol) C, D, E.

By removing the useless symbols, the grammar will be

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

Removal of Null Productions:

A production in the form $NT \rightarrow \epsilon$ is called Null production.

If ϵ (null string) is the language set generated from the grammar, then the null production cannot be removed

That is, if we get $S \overset{*}{\Rightarrow} \epsilon$. then that null production cannot be removed from the production rules

Procedure to remove Null production:-

A procedure in the form of $NT \rightarrow \epsilon$ is called Null production

If ϵ is the string language set generated from the grammar, then the null production can be removed.

That is if we get $\overset{*}{\Rightarrow} \epsilon$, then the null production can't be removed from the production rule.

Procedure to Remove Null production:

Step1 :- If $A \rightarrow \epsilon$ is a production to be eliminated, then we look for all productions whose right side contains A

Step2 :- Replace each occurance of A in each of those productions to obtain the non-ϵ production.

Step3 :- These non-null productions must be added to the grammar to keep the grammar language generating power the same.

1) Remove the ϵ production from the following grammar

$S \rightarrow aA$

$A \rightarrow b \mid \epsilon$

Sol:- In the previous production rules, there is a null production $A \rightarrow \epsilon$. the grammar doesn't produce null string as a language & so this null production can be removed

According to the first step we have to look for all productions, whose right side contains ~~there is~~ ~~one step~~ A.

there is one $S \rightarrow aA$

According to the second step, we have to replace each occurance of 'A' in each of the production, there is only one occurance of 'A' in $S \rightarrow aA$. So, after replacing it will become $S \rightarrow a$

According to step 3, these productions will be added to the grammar.

So, the grammar will be

$$S \rightarrow aA \mid a$$
$$A \rightarrow b.$$

Now;

Now, in the new production rules, there is no null production.

Remove the ϵ production from the following Grammar

$$S \rightarrow aX \mid bX$$
$$X \rightarrow a \mid b \mid \epsilon$$

$S \rightarrow aX$

$S \rightarrow bX$

$X \rightarrow a$

$X \rightarrow b$

$X \rightarrow \epsilon$

According to second step. There is only one occurence of $S \rightarrow aX$, $S \rightarrow bX$. After replacing replace 'e' in place of X.

it will become :-

$S \rightarrow a$

$S \rightarrow b.$

The grammar will be

$S \rightarrow aX \mid a$

$S \rightarrow bX \mid b$

$X \rightarrow a$

$X \rightarrow b$

(or)

$S \rightarrow aX \mid bX \mid a \mid b$

$X \rightarrow a \mid b$

Remove the ε- production

$S \longrightarrow ABaC$

$A \longrightarrow BC$

$B \longrightarrow b \mid \epsilon$

$C \longrightarrow D \mid \epsilon$

$D \longrightarrow d$

The productions are: $S \longrightarrow ABaC$

$A \longrightarrow BC$

$B \longrightarrow b$

$B \longrightarrow \epsilon$

$C \longrightarrow D$

$C \longrightarrow \epsilon$

$D \longrightarrow d$

replacing ϵ in place of C & B.

$S \longrightarrow ABa$

$S \longrightarrow Aa$

$S \longrightarrow AaC$

$A \longrightarrow B$

$A \longrightarrow C$

the grammar will be

$S \longrightarrow ABaC \mid ABa \mid AaC \mid Aa$

$A \longrightarrow BC \mid B \mid C$

$C \longrightarrow D$

$D \longrightarrow d$

$B \longrightarrow b$

Remove the ε production: $S \longrightarrow ABAC$

$A \longrightarrow aA \mid \epsilon$

$B \longrightarrow bB \mid \epsilon$

$C \longrightarrow \epsilon$

A) the given productions are:-

$S \longrightarrow ABAC$

$A \longrightarrow aA \; ; A \longrightarrow \epsilon$

$B \longrightarrow bB \; ; B \longrightarrow \epsilon$

$C \longrightarrow C$

replacing 'ε'

$S \longrightarrow aABAC$

$S \longrightarrow BAC$

$S \longrightarrow AbBAC$

$S \longrightarrow AAC$

$S \longrightarrow ABAC$

$S \longrightarrow BAC$

$S \longrightarrow ABC$

$S \longrightarrow AAC$

$S \longrightarrow BC$

$S \longrightarrow C$

$S \longrightarrow AC$

Linear Grammar:-

The grammar will be

$$S \longrightarrow BAC \mid ABC \mid AAC \mid BC \mid C \mid AC \mid ABAC$$

$$A \longrightarrow aA$$

$$B \longrightarrow bB$$

$$C \longrightarrow c$$