UNIT - III

Context Free Grammar

Introduction

Context free grammar, in(front) short CFG, is type 2 grammar according to the chomsky hierarchy. The Language generated by the CFG is called context-free language (CFL). Regular grammar is a subset of CFG.

Definition of context-free Grammar

→ In mathematical description, we can describe it as where all the Production are in the form $\alpha \rightarrow \beta$

where $\alpha \in V_N$, that is set of non-terminals and $|\alpha| = 1$, that is there will be only one non-terminal at the left hand side (LHS).

where $\beta \in V_N \cup \Sigma$, β is a combination of non-terminals and terminals.

→ The Production rules are in the format of

{A string consists of at least one non-terminal} →

{A string of terminals and/or non terminals}.

→ If any symbol is Present with the Producing non-terminal at the LHS of the Production rule, then that extra symbol is called context.

Context can be of two types: a) Left context

b) Right context

→ In CFG, at the LHS of each production rules, there is only one non-terminal (No context is added with it). For this reason, this type of grammar is called CFG.

Examples

① construct a CFG for the language $L = \{WCW^R / W \in (a,b)^*\}$

Sol: W is a string of any combination of a and 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. Therefore, if we take C as a mirror, we will be able to see the reflection of W in the W^R part, for instance, C, abCba, abbaCabba, and so on. That means, there is something that is generating symbols by replacing which it adds the same symbol before C and after C.

As $W \in (a,b)^*$, the null symbols are also accepted in the place of a, b. That means, only C is accepted by this language set.

From the previous discussion, the production rules are

$$S \rightarrow aSa / bSb / C$$

The grammar becomes

$$G = \{V_N, \Sigma, P, S\}$$

where $V_N : \{S\}$

$\Sigma : \{a, b, C\}$

$P : S \rightarrow aSa / bSb / C$

$S : \{S\}$

Derivation and Parse Tree

In the Process of generating a language from the given Production rules of a grammar, the non-terminals are replaced by the corresponding strings of the right hand side (RHS) of the Production. But if there are more than one non-terminal, then which of the ones will be replaced must be determined.

Depending on this selection, the derivation is divided into two Parts:

1) Leftmost derivation: A derivation is called a leftmost derivation if we replace only the leftmost non-terminal by some production rule at each step of the generating process of the language from the grammar.

2) Right most derivation: A derivation is called a right-most derivation if we replace only the right-most non-terminal by some production rule at each step of the generating process of the language from the grammar.

Example

Construct the string 0100110 from the following grammar by using i) left-most derivation ii) Right most derivation.

$$S \to 0S \,|\, 1AA$$
$$A \to 0 \,|\, 1A \,|\, 0B$$
$$B \to 1 \,|\, 0BB$$

Sol: i) Leftmost Derivation

$S \to 0\underline{S} \to 01\underline{A}A \to 010\underline{B}A \to 0100\underline{B}BA \to 01001\underline{B}A \to 010011\underline{A}$
$\to 0100110$

(The non-terminals that are replaced are underlined).

ii) Right most Derivation

$S \to 0\underline{S} \to 01A\underline{A} \to 01\underline{A}0 \to 010\underline{B}0 \to 0100B\underline{B}0 \to 0100\underline{B}10$
$\to 0100110$

(The non-terminals that are replaced are underlined).

Parse Tree

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

→ A Parse tree is the tree representation of deriving a CFL from a given content grammar.

→ A Parse tree is an ordered tree in which the LHS of a production represents a Parent mode and the RHS of a production represents a children mode.

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

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

LMD: $S \rightarrow 0\underline{S} \rightarrow 01\underline{A}A \rightarrow 010\underline{B}A \rightarrow 0100\underline{B}BA \rightarrow 01001\underline{B}A \rightarrow 0100110$

RMD: $S \rightarrow 0\underline{S} \rightarrow 01A\underline{A} \rightarrow 01\underline{A}0 \rightarrow 010B0 \rightarrow 0100B\underline{B}0 \rightarrow 0100\underline{B}10 \rightarrow 0100110$

For this derivation, the Parse tree is

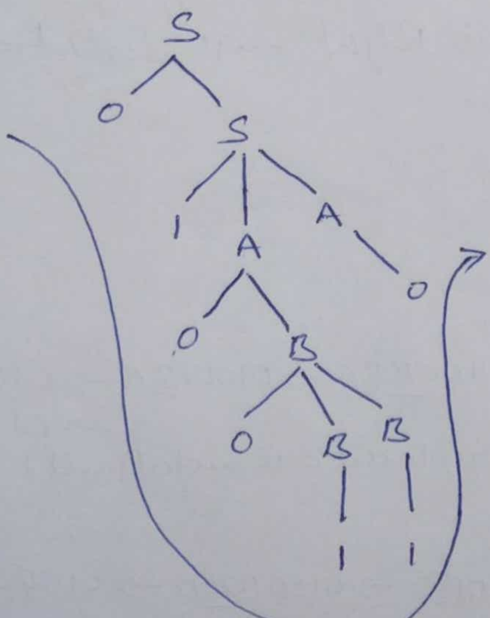


Fig: Parse tree for LMD

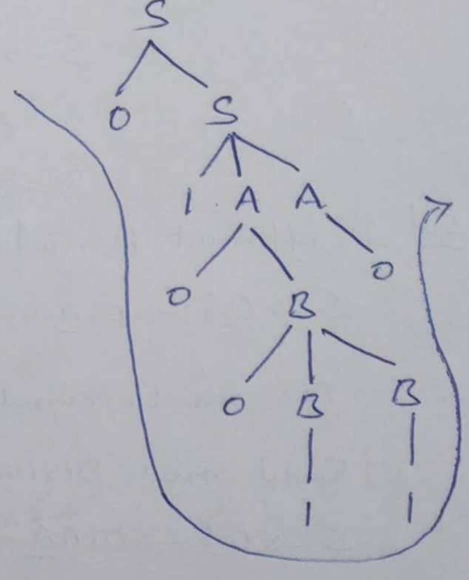


fig: Parse tree for RMD

Ambiguity in CFG

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

→ We know ~~to~~ two types of derivations : i) leftmost derivation ii) right most derivation. Except these two, there is also another approach called the mixed approach means whereas any of the non-terminals present in the deriving string is replaced by a suitable production rule.

① Check whether the grammar is ambiguous or not

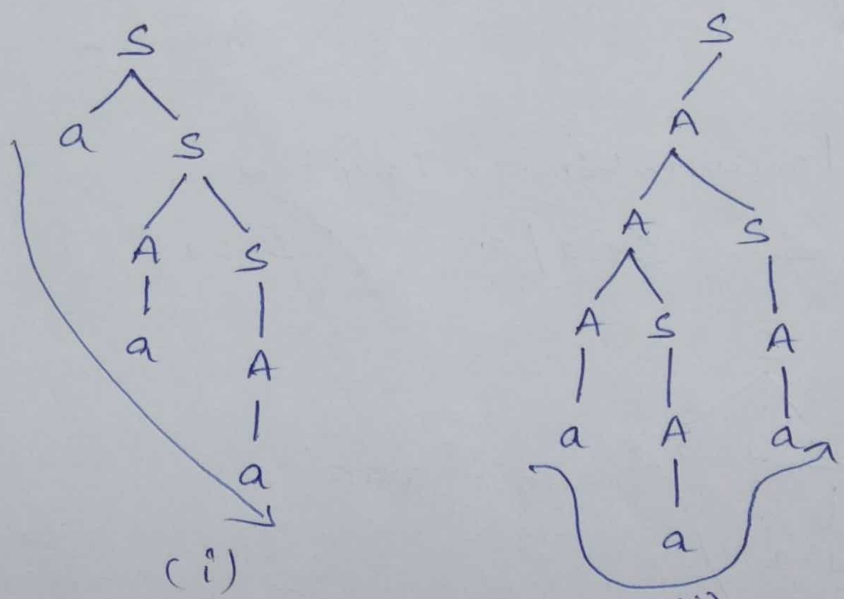
$$S \rightarrow aS \mid AS \mid A$$
$$A \rightarrow AS \mid a$$

Sol:- Consider the string 'aaa'. The string can be generated in many ways. Here, we are giving two ways.

i) $S \rightarrow a\underline{S} \rightarrow a\underline{A}S \rightarrow aa\underline{S} \rightarrow aa\underline{A} \rightarrow aaa$

ii) $S \rightarrow A \rightarrow \underline{A}S \rightarrow \underline{A}\underline{S}S \rightarrow a\underline{A}S \rightarrow aa\underline{S} \rightarrow aaA \rightarrow aaa$

The Parse trees for derivation (i) and (ii)



(i) (ii)

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

Removing Ambiguity

A grammar of a language is called unambiguous if any of the cases for generating a particular string, only one parse tree can be generated.

(i) Associativity Problem

Ex: $w : ((a+a)+a)$ (or)

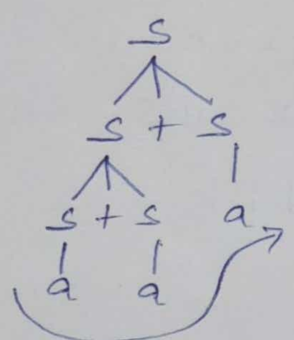
$w : (a+(a+a))$

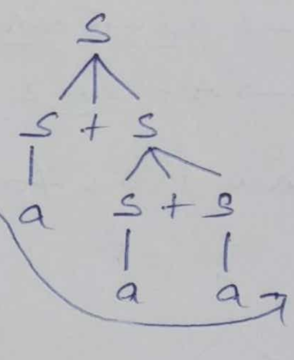
If grammar has left association operator $(+, -, *, /)$ then ~~...~~ induce the left recursion.

If grammar has right association operation $(+, -, *, /)$ then induce the right recursion.

$S \rightarrow S+S / a$

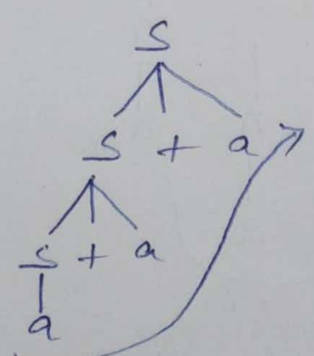
Left recursion | right recursion





To generate unambiguous parse tree

$S \rightarrow S+a / a \quad (\therefore S \rightarrow a)$



---> 'aaa'

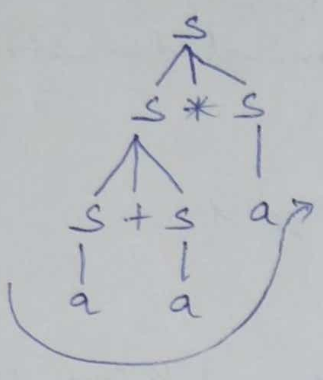
There is only one way to generate 'aaa' string. So according to the definition, the grammar is unambiguous.

ii) Precedence Problem:

Ex: w: a+a*a

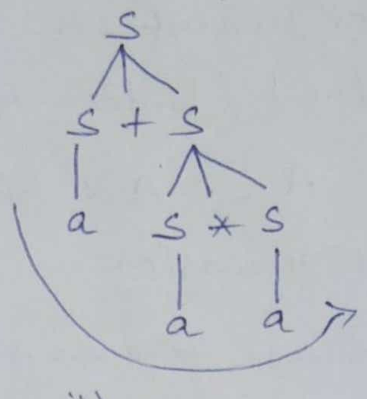
$S \rightarrow S+S \mid S*S \mid a$

Left recursion



(i) a+a*a

Right recursion



ii) a+a*a.

To generate unambiguous grammar.

$S \rightarrow S+T$ (by $S \rightarrow T$) $\therefore S \rightarrow S+S$

$S \rightarrow T$

$T \rightarrow T*F$ (by $T \rightarrow F$) $\therefore S \rightarrow S*S$

$T \rightarrow F$

$F \rightarrow a$ (by $S \rightarrow T \rightarrow F \rightarrow a$) $\therefore S \rightarrow a$

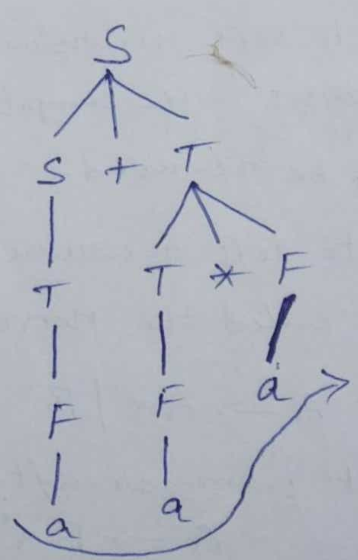
$\Rightarrow S \rightarrow S+S \mid S*S \mid a$

⇓

$S \rightarrow S+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow a$



$\therefore$ a+a*a => unambiguous

Left recursion & Left Factoring

A context free grammar is called left recursive if a non-terminal 'A' as a left most symbol appears alternatively at the time of derivation either immediately (called direct left recursion) or through some other non-terminal definitions (called indirect / hidden left recursive).

$$A \overset{+}{\Rightarrow} A\alpha \text{ for some string } \alpha$$

Direct left recursion

Let the grammar be $A \rightarrow A\alpha / \beta$, when α and β consists of terminal and/or non-terminals but β does not start with A.

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

Indirect left recursion:

Take a grammar in the form

$$\begin{array}{l} A \rightarrow B\alpha / \gamma \\ B \rightarrow A\beta / \delta \end{array} \Rightarrow \begin{array}{l} A \rightarrow \underline{B}\alpha \\ A \rightarrow A\beta\alpha \text{ (by } B \rightarrow A\beta) \end{array}$$

Remove left Recursion

Immediate left recursion creates problems in top down parsing of syntax analysis in the compiler design. So, immediate left recursion must be removed.

Immediate left recursion can be removed by the following process. This is called the Moore's proposal. For a grammar in the form. $A \rightarrow A\alpha / \beta$, where β does not start with A, the equivalent grammar after removing left recursion becomes

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

1. Remove the left recursion from the following grammar.

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

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

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

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

The left recursion $T \rightarrow T*F \mid F$ is removed as

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

The CFG after removing the left recursion becomes

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow id \mid (E)$$

Left Factoring

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

After left factoring, the previous grammar is transformed into:

$$A \rightarrow \alpha A_1$$
$$A_1 \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

① Left Factor the following grammar.

$$A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB$$

Sol: In the above grammar, the RHS Productions abB and aB both start with 'a'. So, they can be left factored. In the same way, cdg, cdeB, and cdfB all start with 'cd'. So, they can also be left factored.

The left factored grammar is

$$A \rightarrow aA'$$
$$A' \rightarrow bB \mid B$$
$$A \rightarrow cdA''$$
$$A'' \rightarrow g \mid eB \mid fB.$$

Simplication of Context free Grammar

CFG may contain different types of useless symbols, unit Productions, and null productions. 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 generating process becomes less from the reduced grammar.

CFG can be simplified in the following three processes.

1) Removal of useless Symbols
   a) Removal of non-generating symbols
   b) Removal of non-reachable symbols
2) Removal of unit Productions
3) Removal of null Productions.

Removal of Useless Symbols

Useless symbols are of two types.

1. Non-generating symbols are those symbols which do not Produce any terminal string.
2. Nonreachable symbols are those symbols which cannot be reached at any time starting from the start symbol.

Example

① Remove the useless symbols from the given CFG.

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

Sol: first, we are finding non-generating symbols.

Those symbols which do not produce any terminal string are non-generating symbols.

Here, C is a 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 (LHS or RHS) must be removed. By Removing the Productions, the minimized grammar will be

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

Now, we have to find non-reachable symbols, the symbols which can not be reached at any time starting from the start symbol. There is no non-reachable symbol in the grammar. So, the minimized form of the grammar by removing useless symbols is

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

Removal of unit productions

Production in the form non terminal $\rightarrow$ 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

$$S \rightarrow \underline{A}B \rightarrow \underline{E}B \rightarrow a\underline{B} \rightarrow a\underline{C} \rightarrow a\underline{D} \rightarrow ab$$

The number of steps is 6.

The grammar, by removing unit productions 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 $S \rightarrow AB \rightarrow aB \rightarrow ab$

The number of steps is 3.

Removal of Null Productions

A Production in the form $NT \rightarrow \epsilon$ is called null Production.

Procedure to remove Null production:

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

Step II :- Replace each occurrence of A in each of these Productions to obtain the non-ϵ production.

Step III :- These non-null productions must be added to the grammar to keep the Language generating power the same.

Example

① Remove the ϵ production from the following grammar

i) $S \to aA$
$A \to b/\epsilon$
by replacing $A \to \epsilon$
$S \to aA | a$
$A \to b$

ii) $S \to aX | bX$
$X \to a|b|\epsilon$
by replacing $X \to \epsilon$
$S \to aX | bX | a | b$
$X \to a|b$

Linear Grammar

A grammar is called linear grammar if it is context free, and the RHS of all productions have at most one non-terminal. As an example, $S \to abSc | \epsilon$, the grammar of $(ab)^n c^n, n \geq 0$

There are two types of linear grammar:

1. Left linear grammar: A linear grammar is called left linear if the RHS non-terminal in each productions are at the left end. In a linear grammar if all productions are in the form $A \to B\alpha$ or $A \to \alpha$, then that grammar is called left linear grammar. Here, A and B are non-terminals and α is a string of terminals.
2. Right linear grammar: A linear grammar is called right linear if the RHS non-terminal in each productions are at the right end. In a grammar if all productions are in the form $A \to \alpha B$ or $A \to \alpha$, then that grammar is called right linear grammar. Here A and B are non-terminals and α is a string of terminals.

Example

① Convert the following linear grammar into Regular Grammar

$S \to baS | aA$
$A \to bbA | bb$

Sol: Consider two non-terminals B and C with production $B \to aS$ and $C \to bA$. The grammar becomes

$S \to bB | aA$
$A \to bC | bb$
$B \to aS$
$C \to bA$

Still the production $A \to bb$ is not a regular grammar. Replace b by a non-terminal D with production $D \to b$. The grammar becomes

$S \to bB | aA$
$A \to bC | bD$
$B \to aS$
$C \to bA$
$D \to b$

Now the grammar is regular.

Normal Form

For a grammar, the RHS of a production can be any string of variables and terminals, i.e., $(V_N \cup \Sigma)^*$. A grammar is said to be in normal form when every production of the grammar has some specific form.

When a grammar is made in normal form, every production of the grammar is converted in some specific form. These helps us to design some algorithm to answer certain questions,

→ Such as if a CFG is converted into Chomsky normal Form (CNF), one can easily answer whether a particular string is generated by the grammar or not.

→ if a CFG is converted into Greibach normal form (GNF), then a PDA accepting the language generated by the grammar can easily be designed.

we have two types of normal forms:

a) CNF (Chomsky Normal Form)

b) GNF (Greibach Normal Form)

Chomsky Normal Form

A CFG is said to be in CNF if all the productions of the grammar are in the following form.

Non-terminal → string of exactly two non-terminals

Non-terminal → Single terminal

A CFG can be converted into CNF by the following process.

Step I :- 1. Eliminate all the ϵ-production

2. Eliminate all the unit production

3. Eliminate all the useless symbols.

Step II :- 1. if (all the productions are in the form

NT → string of exactly two NTs

(or)

NT → Single terminal

Declare the CFG is in CNF and stop.

2. else (follow step III and/or IV and/or V).

Step III :- Elimination of terminals on the RHS of length two or more.

Step IV :- Restriction of the number of variables on the RHS to two.

Step V :- conversion of the string containing terminals and non-terminals to the string of non-terminal on the RHS.

Example

① convert the following grammar into CNF.

$S \to bA \mid aB$

$A \to bAA \mid aS \mid a$

$B \to aBB \mid bS \mid a$

Sol :- The productions $S \rightarrow bA$
$S \rightarrow aB$
$A \rightarrow bAA$
$A \rightarrow aS$
$B \rightarrow aBB$
$B \rightarrow bS$
are not in CNF.

So, we have to convert these into CNF.

$C_a \rightarrow a$
$C_b \rightarrow b$

By Replacing a and b by new non-terminals and including the two productions, the modified grammar will be

$S \rightarrow C_bA$
$S\cancel{A} \rightarrow C_aB$
$A \rightarrow C_bAA \mid C_aS \mid a$
$B \rightarrow C_aBB \mid C_bS \mid a$
$C_a \rightarrow a$
$C_b \rightarrow b$

Two new productions will be added to the grammar

$D \rightarrow AA$ and $E \rightarrow BB$.

So, the new modified grammar will be

$S \rightarrow C_bA \mid C_aB$
$A \rightarrow C_bD \mid C_aS \mid a$
$B \rightarrow C_aE \mid C_bS \mid a$
$D \rightarrow AA$
$E \rightarrow BB$
$C_a \rightarrow a$
$C_b \rightarrow b$

(2) Convert the following grammar into CNF.

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

Greibach Normal Form

A grammar is said to be in GNF if every production of the grammar is of the form.

Non-terminal → (Single terminal)(string of NTs)

Non-terminal → Single terminal.

In one line, it can be said that all the productions will be in the form

Non-terminal → (Single terminal)(non-terminal)*

Step I :- check if the given CFG has any unit productions or null productions and remove if there are any.

Step II :- Check whether the CFG is already in CNF and convert it to CNF if it is not.

Step III :- Change the names of the NT symbols into some A_i in ascending order of i.

Step IV :- Alter the rules so that the NT's are in ascending order such that if the production is of the form $A_i \to A_j^*$ then $i<j$ and should never be $i \geq j$.

Step V :- If there is no combination of single terminal with string of NT's, we can apply "remove left recursion".

Example

(1) Convert the following grammar into GNF

$S \to CA \mid BB$

$B \to b \mid SB$

$C \to b$

$A \to a$

Sol :

Step I :- There are no unit productions and no null productions in the grammar.

Step II :- The given grammar is in CNF.

Step III :- change the names of the NT symbols into some A_i in ascending order of i.

$S \to CA \mid BB$

$B \to b \mid SB$

$C \to b$

$A \to a$

Rename the non-terminals as

$S \to A_1$

$C \to A_2$

$A \to A_3$

$B \to A_4$

then

$A_1 \to A_2A_3 \mid A_4A_4$

$A_4 \to b \mid A_1A_4$

$A_2 \to b$

$A_3 \to a$

Step IV :- After the rules, the non-terminals are in ascending order or not.

$A_4 \to b \mid A_1A_4$ not in ascending order.

$A_4 \to b \mid A_2A_3A_4 \mid A_4A_4A_4$ ($\because A_1 \to A_2A_3 \mid A_4A_4$)

$A_4 \to b \mid bA_3A_4 \mid A_4A_4A_4$ (by $A_2 \to b$)

Step V :- There is no combination of single terminal with string of NT's, we can apply "remove left recursion).

$A_4 \to b \mid bA_3A_4 \mid \underline{A_4A_4A_4}$ → Apply left recursion

$Z \to A_4A_4Z \mid A_4A_4$

$A_4 \to b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

Now the grammar is

$A_1 \to A_2A_3 \mid A_4A_4$

$A_4 \to b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

$Z \to A_4A_4Z \mid A_4A_4$

$A_2 \to b$

$A_3 \to a$

In GNF,

$A_1 \rightarrow bA_3 \mid A_4 A_4$ (by $A_2 \rightarrow b$)

$A_1 \rightarrow bA_3 \mid bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4$

(replaced by A_4)

$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

$Z \rightarrow bA_4Z \mid bA_3A_4A_4Z \mid bZA_4Z \mid bA_3A_4ZA_4Z \mid bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$.

Closure Properties of Context-Free Language

→ A set is closed (under an operation) if and only if the operation on two elements of the set produces another element of the set. If an element outside the set is produced, then the operation is not closed.

→ closure is a property which describes the application of the property on any two elements of the set; the result is also included in the set.

i) closed Under Union:

$L = L_1 \cup L_2$ is also in CFL

Let us construct a grammar $G = (V_N, \Sigma, P, S)$ using the two grammars G_1 and G_2 as follows

$V_N = V_{N_1} \cup V_{N_2} \cup \{S\}$

$\Sigma = \Sigma_1 \cup \Sigma_2$.

$P = P_1 \cup P_2 \cup \{S \rightarrow S_1 / S_2\}$

ii) closed under concatenation:

$L = L_1 L_2$ is in CFL

$V_N = V_{N_1} \cup V_{N_2} \cup \{S\}$

$\Sigma = \Sigma_1 \cup \Sigma_2$

$P = P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}$

iii) closed under star closure :

L^* is in CFL $\quad V_N = V_{N_1} \cup \{S\}$

$P = S \rightarrow S_1 S / \lambda$

iv) closed under Intersection:

$L = L_1 \wedge L_2$

so, $L = \underbrace{a^{n+1} b^{n+1} c^n}_{L_1} \wedge \underbrace{a^n b^n c^{n+k}}_{L_2} = a^n b^n c^n$, when $n \geq 0$

$a^n b^n c^n$ is a context sensitive language not a context free.

v) Not closed under complementation :

$L_1 \wedge L_2 = L_1 \cup L_2$. (D' Morgan's Law)

we are getting contradiction here, so, CFLs are not closed under complementation.

~~Decision Problems~~

Pumping Lemma for CFL

we have become familiar with the term 'pumping lemma' in the regular expression chapter. The pumping lemma is also related to a CFL.

→ The pumping lemma for CFL is used to prove that certain sets are not context free.

→ Every CFL fulfills some general properties. But if a set or language fulfills all the properties of the pumping lemma for CFL, it cannot be said that the language is not context free.

Pumping Lemma for CFL: Let L be a CFL. Then, we can find a natural number n such that

1. Every $z \in L$ with $z \geq n$ can be written as $\omega = uv\omega xy$,
2. $|vx| \geq 1$
3. $|v\omega x| \leq n$
4. $uv^k \omega x^k y \in L$ for all $k \geq 0$

Pumping Lemma (for CFL) is used to prove that a Language is NOT Content Free.

If L is a Content Free Language, then L has a Pumping Length 'P' such that any string 's', where $|S| \geq P$ may be divided into 5 pieces $S = uvxyz$ such that the following conditions must be true:

(1) uv^ixy^iz is in L for every $i \geq 0$

(2) $|vy| > 0$

(3) $|vxy| \leq P$

Example

① Show that L is not Content Free

$$L = \{a^N b^N c^N / N \geq 0\}$$

sol :- → Assume that L is content free

→ L must have a pumping length (say P)

→ Now we take a string S such that $S = a^P b^P c^P$

→ we divide S into parts $uvxyz \in L$

Pumping Length $P = 4$

So, $S = a^4 b^4 c^4$

Case 1: v and y each contain only one type of symbol

$$S = a^4b^4c^4 = \underbrace{a}_{u}\underbrace{aa}_{v}\underbrace{abbbbc}_{x}\underbrace{c}_{y}\underbrace{cc}_{z}$$

i) uv^ixy^iz $(i = 2)$

uv^2xy^2z = a aaaa abbbbccccc

$a^6 b^4 c^5 \notin L$

ii) $|vy| > 0 \Rightarrow 3 > 0$, iii) $|vxy| \leq P \Rightarrow 9 \leq 4 \notin L$

Case 2: Either v or y has more than one kind of symbols.

$S = uvxyz$

$S = a^4 b^4 c^4 = \underbrace{aa}_{u}\underbrace{aabb}_{v}\underbrace{b}_{x}\underbrace{b}_{y}\underbrace{cccc}_{z}$

(i) $uv^i xy^i z$ $(i=2)$

$uv^2xy^2z \Rightarrow aaaabbaabbbbbcccc \notin L$

ii) $|vy| > 0 \Rightarrow 5 > 0$

iii) $|vxy| \leq P \Rightarrow 6 \leq 4 \notin L$

Hence our assumption of L being CFG is wrong.

This Process that given language L is not Context Free.