Module-3 Ch-1

Context free grammar and Context free languages

Objectives

1) Context free Grammar (CFG)
   - i) what is CFG.
   - ii) what is the use.
   - iii) Examples.
   - iv) leftmost derivation & Rightmost derivation
   - v) parse tree / Derivation tree.
2) Ambiguity in CFG.
   - what is Ambiguity.
   - why it occures
   - Problems.
   - Examples
3) minimization of CFG.
   - a) Removal of useless symbols.
     - Non generating symbols.
     - Non reachable symbols
   - b) Removal of unit production.
   - c) Removal of null production.
4) Chomsky Normal Form. (CNF)
5) Greibach Normal Form (GNF)
6) Enumeration of properties of CFL. (without any proves)
7) closure property of CFL.
8) ogden's lemma & its application.

Objective-1

Context Free grammar (CFG)

Definition of CFG.

A grammar $G = (N, T, P, S)$ is said to be context free if all productions in P are in the following form.

$$A \rightarrow x.$$

where $A \in N$ (i.e. set of all nonterminals)

$x \in (N \cup T)^*$

P is a set of all production rules.

$S \in N$ is a starting symbol.

Example of CFG.

A grammar $G = (\{S\}, \{a, b\}, \{P\}, S)$

with production, $S \rightarrow aSa$

$S \rightarrow bSb$

$S \rightarrow \epsilon$

is context free.

A typical derivation of this grammar is,

$S \rightarrow aSa$

$S \rightarrow aaSaa$ [$S \rightarrow aSa$]

$S \rightarrow aabSbaa$ [$S \rightarrow bSb$]

$S \rightarrow aabbaa.$ [$S \rightarrow \epsilon$]

Note: CFGs derives their name from the fact, that substitution of the variable on the leftmost of a production can be made any time. such a variable appears in a sentinal form. It doesnot depend on the symbols in the rest of the Sentinal form (context). This feature is the consequence of allowing only a single variable on the leftside of the production.

Leftmost & Rightmost Derivation.

1) A CFG may not be linear
2) A derivation may involve a sentinal form with more than one variable.
3) In such cases (more than 1 non terminal in right side) we have to choose which variable to replace.
4) The order of replacement specifies the leftmost or rightmost derivation.
(This replacement should be recursive in nature)

Problem.

A grammar $G = (\{A, B, S\}, \{a, b\}, P, S)$
with productions, 1) $S \to aAB$
2) $A \to aaA$
3) $A \to \epsilon$
4) $B \to Bb$
5) $B \to \epsilon$

i) Calculate the leftmost derivation for the above grammar G.
ii) Calculate the rightmost derivation for above grammar G.
iii) state the language for the grammar G.

---

i) $S \to \underline{A}B$ (1)
$S \to aa\underline{A}B$ (2)
$S \to aa\underline{B}$ (3)
$S \to aa\underline{B}b$ (4)
$S \to aab$ (5)

ii) $S \to A\underline{B}$ (1)
$S \to A\underline{B}b$ (4)
$S \to \underline{A}b$ (5)
$S \to aa\underline{A}b$ (2)
$S \to aab$ (3)

iii) Language.

Let us assume b will be repeated m times & a will be repeated n times

The condition is, $n = 2m$.

∴ Language is $L(G) = \{ a^n b^m \mid n \geq 0 \; m \geq 0 \}$

$= \{ a^{2m} b^m \mid m \geq 0 \}$

Derivation/Parse Tree.

What is the use of Derivation/Parse tree?

— With the help of parse tree/derivation tree, we can show the derivations independent of the order in which productions are used.

What do you mean by leftmost derivation?

— A derivation is called a leftmost derivation if we replace only the leftmost terminal by some production rule at each step of the generating process of the language from the grammar.

What do you mean by rightmost derivation?

— A derivation is said to be rightmost, iff we replace only the rightmost nonterminal by some production rule at each step of the generating process of the language from the grammar

What is parsing?

— Parsing a string means finding a derivation for that string from a given grammar.

Parse tree / Derivation tree

A parse tree is an ordered tree in which lefthand side of a production represents a parent node and children nodes are represented by the production's right hand side.

What are the conditions for constructing a parse tree from a CFG.?

i) Each vertex of the tree must have a label. The label is a terminal or non-terminal or null(ε).

ii) The root of the tree is the start symbol i.e. (S)

iii) Label of the internal vertices are, non-terminal symbols belongs to V_N (set of all vertices)

iv) 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 that node will be $X_1, X_2, X_3 \ldots X_k$.

v) A vertex n. is called a leaf of the parse tree if its label is a terminal symbol belongs to Σ or ε

Why parse tree construction is only possible for CFG.?

Parse tree construction is only possible for CFG. This is because the properties of a tree match with the properties of a CFG. Here we are describing the similar properties of a tree & a CFG.

1) For a tree there must have a root. Similarly for every CFG there is a single starting symbol.

2) Each node a tree has a single label.
For every CFG at the left hand side of each production, there is a single non production

3) A child node is derived from a single parent
For constructing a CFL for a given CFG. a non terminal is replaced by a suitable string at the right hand side. Each of the character of the string is generating a node. i.e. to say for each single node there is a parent

problem-1

Find the parse tree for generating the string 0100110 from the grammar given below,

$S \rightarrow 0S / 1AA$ ..(i)

$A \rightarrow 0 / 1A / 0B$ ..(ii)

$B \rightarrow 1 / 0BB$ ...(iii)

Left

$S \rightarrow 0\underline{S}$ ..(i)

$S \rightarrow 01\underline{A}A$ ... (i)

$S \rightarrow 010\underline{B}A$ ...(ii)

$S \rightarrow 0100\underline{B}BA$ ..(iii)

$S \rightarrow 01001\underline{B}A$ ...(iii)

$S \rightarrow 0100110$ ...(ii)

Right

$S \rightarrow 0\underline{S}$ ..(i)

$S \rightarrow 01A\underline{A}$ ..(i)

$S \rightarrow 01A0\underline{B}$ ...(ii) [A→0]

$S \rightarrow 010\underline{B}0B$ ...(ii) [B→0B]

$S \rightarrow 0100B\underline{B}0B$ ..(iii) [B→0BB]

$S \rightarrow 0100\underline{B}10$ ...(iii) [B→1]

$S \rightarrow 0100110$ ...(iii) [B→1]

Right

$S \rightarrow 0S$

$S \rightarrow 01AA$

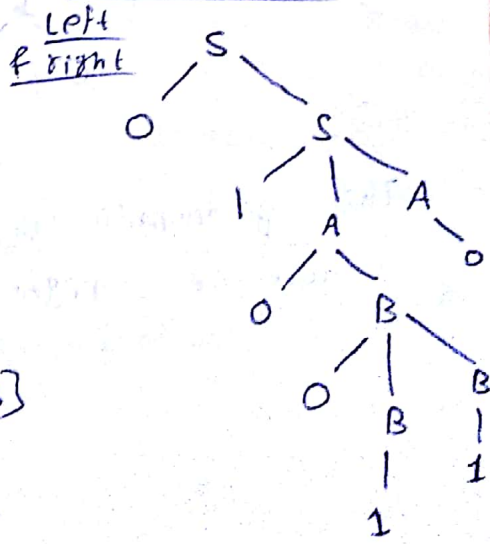
$S \rightarrow 01A0B$

$S \rightarrow 01A11A$

$S \rightarrow 01A110$

parse tree

Left & right



∴ single parse tree for both are same .. it is not ambiguous

Find the parse trees for leftmost and right most derivations for generating the string,

11001010 for the given grammar.

$S \to 1B/0A$

$A \to 1/1S/0AA$

$B \to 0/0S/1BB$

Left

$S \to 1\underline{B}$

$S \to 11\underline{B}B$ [B → 1BB]

$S \to 110\underline{S}B$ [B → 0S]

$S \to 1100\underline{A}B$ [S → 0A]

$S \to 11001\underline{S}B$ [A → 1S]

$S \to 1100101\underline{A}B$ [S → 0A]

$S \to 11001010\underline{B}$ [A → 1]

$S \to 11001010$ [B → 0]

Right

$S \to 1\underline{B}$

$S \to 11\underline{B}\underline{B}$ [B → 1BB]

$S \to 11\underline{B}0$ [B → 0S]

$S \to 110\underline{S}0$ [B → 0S]

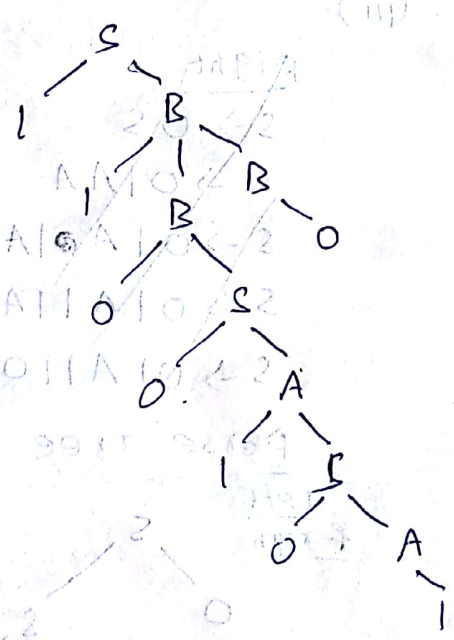
$S \to 1100\underline{A}0$ [S → 0A]

$S \to 11001\underline{S}0$ [A → 1S]

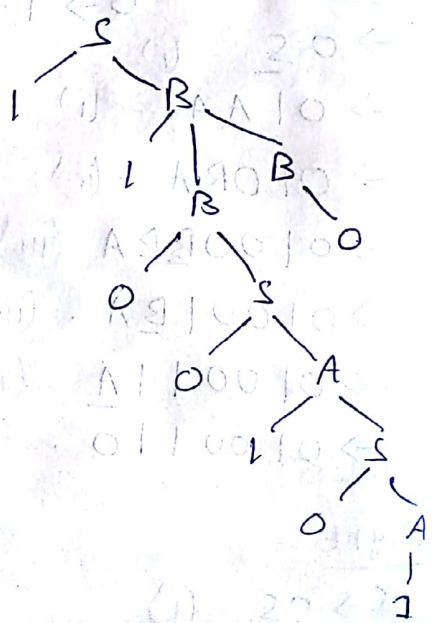
$S \to 1100101\underline{A}0$ [S → 0A]

$S \to 11001010$ [A → 1]

Tree (Left)



(Right)



This grammar is ambiguous as it has multiple right/leftmost derivations.

Construct a parse tree for the string aabbaa from the grammar given below.

S → a / aAS

A → SS / SbA / bba

Left

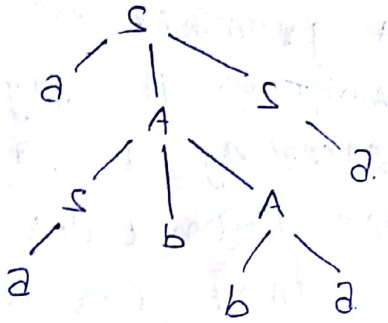
S → aAS

S → aSbAS [A → SbA]

S → aabAS [S → a]

S → aabbaS [A → ba]

S → aabbaa [S → a]



Construct a string abbbb from the grammar given below

a) Left most derivative

b) Right most derivative

S → aAB

A → bBb

B → A / ε

Left:

S → aAB

S → AbBbB

S → AbbB

S → Abb

S → aAB

S → abBbB

S → abAbB

S → abbBbbB

S → abbbbB

S → abbbbb

Right

S → aAB

S → aAA

S → aAbBb

S → aAbb

S → abBbbb

S → abbbb

Ambiguity

what do you mean by ambiguous language or ambiguous grammar? Explain with example.

1) A grammar of a language is called ambiguous if any of the cases for generating a particular string, more than one leftmost derivation or more than one rightmost derivation or more than one parse tree can be generated.

2) Ambiguous Language

i) Ambiguous language is generated from an ambiguous grammar.

ii) An ambiguous language can be generated using either of the following cases:-

a) It is generated through more than one leftmost derivations

b) It is generated through more than one rightmost derivations.

c) It is generated through more than one parsetree / derivation tree.

3) Example

Let us assume there is a grammar

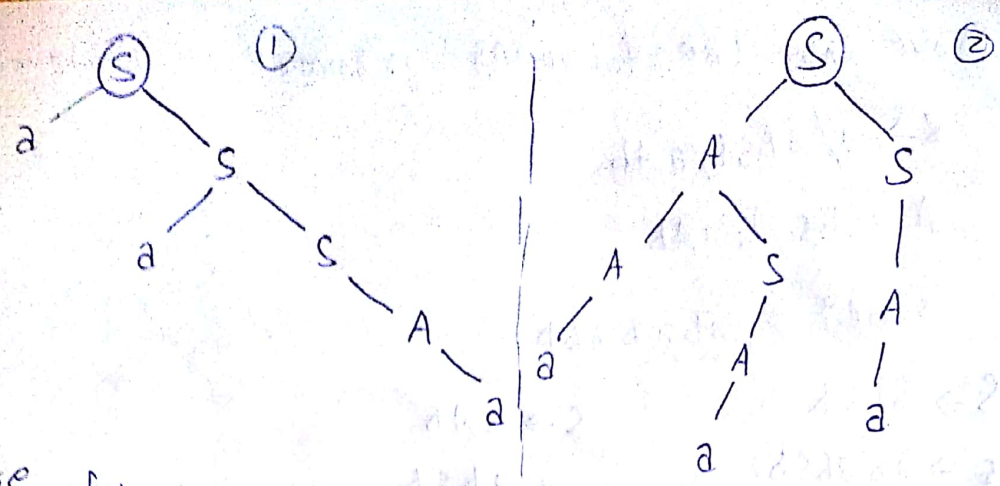
$S \to aS / AS / A$

$A \to AS / a.$

Now generate the string aaa from the grammar.

① $S \to aS \to aaS \to aaA \to aaa$
[S→aS] [S→A] [A→a]

② $S \to AS \to ASS \to aSS \to aAS \to aaS \to aaA \to aaa$
[A→AS] [A→a] [S→A] [A→a] [S→A] [A→a]



Here for the same string derived from the same grammar, we are getting more than one leftmost derivations aswell as more than passe tree, so the grammar is ambiguous.

prove that the following grammar is ambiguous

V_N : {S} → set of set variables

Σ : {id, +, *} → set of all terminals

P : S → S+S / S*S / id. → production rules

s : {S}

String

id + id * id.

S → S+S

→ id +S

→ id +S*S

→ id + id *S

→ id + id* id.

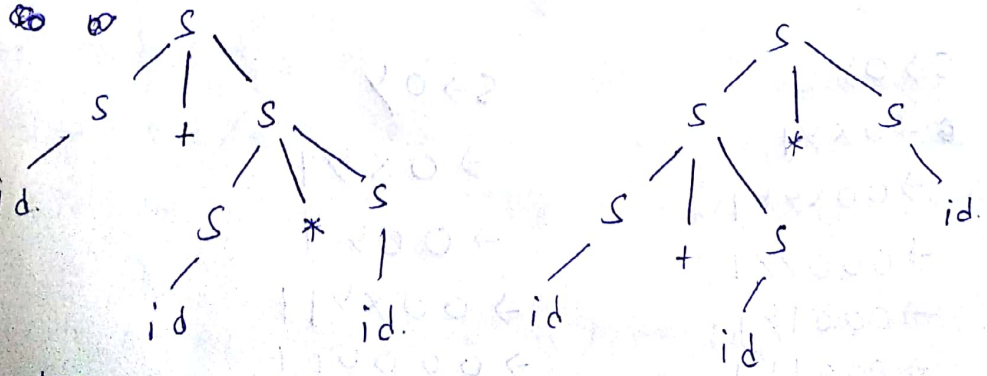
S → S*S

S → S+S*S

S → id+S *S

S → id + id *S

S → id + id * id.



∴ two different left most generation passe trees are possible for same string. ∴ Ambiguous.

Prove that the following grammar is ambiguous

$S \to a / abSb / aAb$

$A \to bS / aAAb$

string $\to$ abababb

$S \to abSb$

$\to ababSbb$

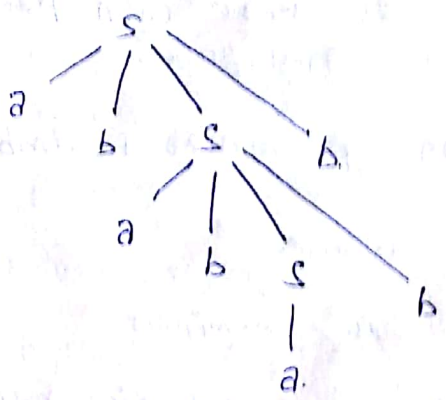
$\to ababbabb$

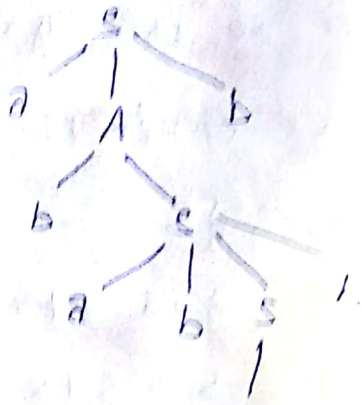
$S \to aAb$

$\to abSb$

$\to ababSbb$

$\to ababbabb$





Since more than one leftmost derivations and parse trees are possible for a single string : ambiguous.

---

prove ambiguous

$S \to 0Y / 01$

$X \to 0XY / 0$

$Y \to XY1 / 1$

Let string is 000111

$S \to 0Y$

$\to 0XY1$

$\to 00XYY1$

$\to 000YY1$

$\to 0001Y1$

$\to 000111$

$S \to 0Y$

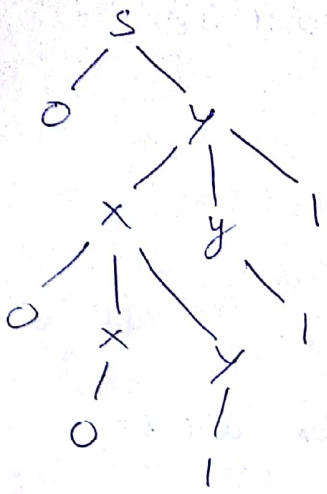
$\to 0XY1$

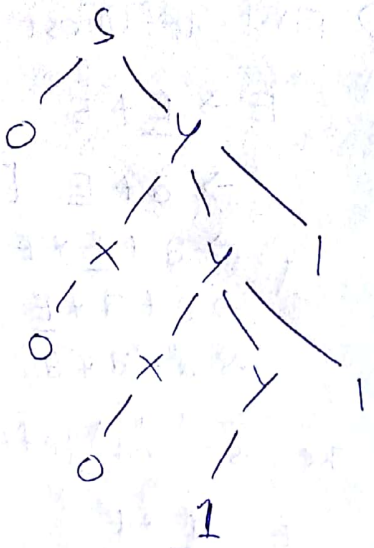
$\to 00Y1$

$\to 00XY11$

$\to 000Y11$

$\to 000111$





∴ The grammar is ambiguous

Define:
Inherently ambiguant CFL

A CFL 'L' is said to be inherently ambiguous if all of its grammars are ambiguous.

Define: Unambiguous CFL.

If 'L' is a CFL, for which there exists an unambiguous grammar 'G', then L is said to be an unambiguous grammar.

Does ambiguous grammar create problem? Explain with a suitable example.

Yes an ambiguous grammar creates problem

We can prove that an ambiguous grammar creates problem with the following example,

Let us assume 'G' be a CFG,

the production rules of this grammar is,

$E \rightarrow E+E / E*E / a$.

we have to construct $a+a*a$. where the value of a is 2.

1) First of all we need to prove that for the grammar 'G' ∃ more than 1 Leftmost derivations.

2) First leftmost derivation will be

$E \rightarrow \underline{E} + E$

$\rightarrow a + \underline{E}$ [E→a]

$\rightarrow a + \underline{E} * E$ [E→E*E]

$\rightarrow a + a * \underline{E}$ [E→a]

$\rightarrow a + a * a.$ [E→a]

3) The second leftmost derivation will be

$E \rightarrow \underline{E} * E$

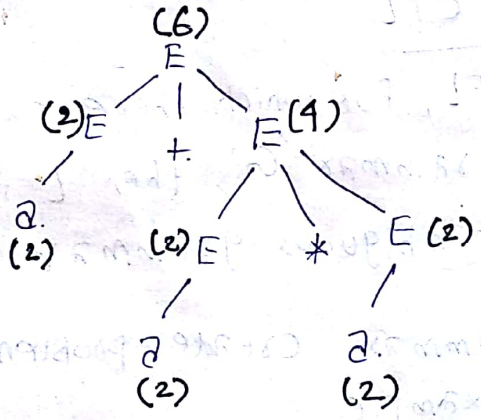
$\rightarrow \underline{E} + E * E$ [E→E+E]

$\rightarrow a + \underline{E} * E$ [E→a]

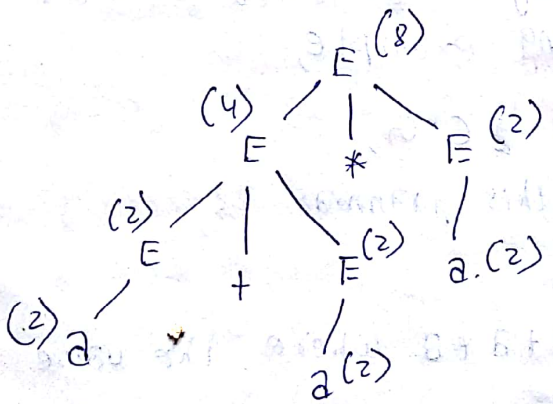
$\rightarrow a + a * \underline{E}$ [E→a]

$\rightarrow a + a * a.$ [E→a]

4) Design the parsetree for the 1st leftmost derivation.



5) Design the parse tree for the 2nd leftmost derivation.



6) We are observing that for the 1st parse tree the value of the root is 6 and for the 2nd parse tree, the value of the root is 8.

but the ... will be 6, as because, according to the operator precedance (*) multiplication operator should be preferred over (+) addition operator, but here according to the parse tree logically we are correct so the computer is confused regarding selection of the parse tree. That leads to undecidibility hence ambiguous grammar creates problem for a computer system.

How can we remove ambiguity from a grammar?

There is no particular rule to remove ambiguity from a CFG.

Though we can apply the following procedures.

1) Ambiguity can be removed by setting priority to the operators.

2) Ambiguity can be removed by changing the production rule (if possible)

3) ∃ some CFL for which ∃ only ambiguous grammar. For this case, the ambiguity cannot be removed.

Left Recursion & Left Factoring

Q1) what is left recursion?

Definition 1) A grammar 'G' is left recursive if it has a non-terminal A, such that there is a derivation as follows,

$$A \xrightarrow{+} A\alpha$$

where 'α' is a string

Definition 2) A CFG is called left recursive if a non terminal 'A' as a leftmost symbol appears alternatively at the time of derivation,

either immediately (called direct left recursive) or throughsome other non terminal definitions (called indirect or hidden left recursive).

$A \rightarrow \underline{A}$

$\rightarrow \underline{A}A$

$\rightarrow \underline{A}AA$

$\rightarrow \underline{A}AAA$

$\rightarrow \ldots\ldots\ldots$

$A \rightarrow \underline{A}\alpha$

$\rightarrow \underline{A}\alpha\alpha$

$\rightarrow \underline{A}\alpha\alpha\alpha$

$\rightarrow A\alpha\alpha\alpha\alpha$

$\ldots\ldots\ldots$

what are the different types of left recursive grammar?

1) Direct left Recursive grammar
2) Hidden / Indirect left Recursive grammar

Direct left Recursive grammar

Let a grammar is $A \rightarrow A\alpha / \beta$

where α & β consist of terminals and / or non terminals.

but β doesnot start with A

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

$A \rightarrow A\alpha$

$A \rightarrow A A \alpha$

$A \rightarrow A\alpha$

$A \rightarrow \beta$.

In direct left recursion.

$A \rightarrow B\alpha / \gamma$

$B \rightarrow A\beta / \delta$

$A \rightarrow \underline{B}\alpha$

$\rightarrow \underline{A}\beta\alpha$

Let us consider a grammar with the following production rule,

$A \rightarrow B\alpha / \gamma$

$B \rightarrow A\beta / \delta$

where α, β, γ & δ consist of terminal or nonterminal

At the time of derivation, if we start from A, and replace A by the production,

$A \rightarrow B\alpha$

and after that the B is replaced by the production $B \rightarrow A\beta$, then A will appear again as a leftmost nonterminal symbol in the RHS of the production.

This kind of recursion is known as indirect left recursion, and this type of grammar is known as indirect left recursive grammar.

Simplification of CFG

1) Why do we need to simplify a CFG?

- CFG may contain different types of useless symbols, unit productions and null productions.

These types of symbols and productions increases number of steps in generating a language from a CFG.

Reduced grammar contains less number of non terminals and production.

So the time complexity for language generating process becomes less for reduced grammar.

2) What are the processes of simplification of a CFG?

- CFG can be simplified in the following 3 processes,

i) Removal of useless symbols

a) Removal of non generating symbols

b) Removal of non reachable symbols

ii) Removal of unit productions

iii) Removal of null production.

3) Define Non generating & Non reachable symbols.

- Non generating symbols

Non generating symbols are those symbols which directly does not produce any terminal string.

Example:- $S \to AC$ $B \to aC$

$S \to BA$ $B \to b$

$C \to CB$

$C \to AC$

$A \to a$.

In this CFG, the symbol that doesnot produce any terminal string are,
S and C, as we donot get any terminal from S and C

Non reachable symbols

Non reachable symbols are those symbols which cannot be reached, at any time starting from the start symbol.

$S \rightarrow bX$
$A \rightarrow bSX / a.$
$X \rightarrow ad.$

In this CFG, the symbol A cannot be reached by any path at any point of time, starting from start symbol S. Hence A is a non reachable symbol.

4) How useless symbols can be removed from a given CFG.

- step 1: Find the non generating symbol in the production rule 'P'. If start symbol is found non generating, then leave that symbol. start symbol cannot be removed as the language generating process start from that symbol.

step 2: For removing non generating symbols remove those productions whose RHS or LHS contain those symbols.

step 3: Find the non reachable symbols from the production rules of the grammar.

step 4: Remove the non reachable symbols, applying the same rule as explained in step 2.

Example

1) Remove useless symbols from the given CFG.

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

Step 1: Here non generating symbols are S and C. as because in RHS of S & C there is only set of non terminals

Step 2: Remove. (Let us delete the productions that contain C in RHS or LHS)

$S \to AC.$
$C \to CB$
$C \to AC.$
$B \to aC.$

Step 3: Here the production rules left after deletion, donot any any non reachable symbols.

∴ there is no non reachable symbol in minimized grammar.

∴ we ignore step 4.

The CFG After eliminating useless symbols is,

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

2) Remove useless symbols from given CFG

$S \rightarrow aB / bX$
$A \rightarrow BAd / bSX / a$
$B \rightarrow aSB / bBX$
$X \rightarrow SBD / aBX / ad.$

step 1: Here non generating symbols are S, B

step 2: Let us delete all production rules containing B on both LHS & RHS

∴ we delete.

~~$S \rightarrow bX$~~ $S \rightarrow aB$
~~$A \rightarrow bSX.$~~ $A \rightarrow BAd.$
~~$B \rightarrow bBX.$~~ $B \rightarrow aSB / bBX.$
~~$X \rightarrow SBD$~~ $X \rightarrow SBD / aBX.$
~~$X \rightarrow aBX$~~
~~$X \rightarrow ad.$~~

Left

$S \rightarrow bX$
$A \rightarrow bSX / a.$
$X \rightarrow ad.$

Step 3: Here the minimized grammar has one non reachable symbol,

A.

step 4.: Let us delete all symbols having A on both LHS & RHS.

∴ we delete.

$A \rightarrow bSX.$
$A \rightarrow a.$

∴ The CFG after eliminating useless symbols is.

$S \rightarrow bX$
$X \rightarrow ad.$

3) $S \to aAa$
$A \to bBB$
$B \to ab$
$C \to aB$

step 1 : Here non generating symbols are
S, A, C.

step 2 : Let us remove all productions containing A & C in both LHS & RHS
we remove
$A \to bBB$
$C \to aB$

step 3 : Here non reachable symbols is
B

step 4 : We remove all productions with B on both LHS & RHS
we remove
$B \to ab$

∴ we get the minimized CFG as
$S \to aAa$.

---

step 1 : First we are going to remove non generating symbols, i.e. which donot produce any terminal string.
Here S, A, C are non generating symbols
As S is the start symbol, so it cannot be removed.
However A & C can be removed.

The production rules containing A or C or both should be removed.

But if we remove A & C from the set of production rules 'p'. Then 'p' becomes $B \to ab$

However there is no meaning of the grammar containing this single production rule and no start symbol.

∴ we have to first remove the non-reachable symbols and then non-generating symbols.

step 2: Here non reachable symbols is C.

So we have to delete the non reachable symbol C from the given grammar.

After deleting C from RHs & LHs of production rule, the grammar is

S → aAa.

A → bBB.

B → ab.

step 3: Now we can remove the non generating symbol S, A.

But if we remove S & A then the grammar will have no meaning.

∴ The production rule remains.

4) Remove the useless symbols from the grammar.

S → aC/SB

A → bSCa/ad

B → ASB/bBC

C → aBC/ad

step 1: Here the non generating symbols are S, B.

step 2: we remove all those productions containg B

we get, S → aC

A → bSCa/ad

C → ad.

Step 3: Here non reachable symbol is A.

Step4: We remove all productions containing A on LHS & RHS

∴ we get the minimized CFG as

$S \rightarrow aC$

$C \rightarrow ad.$

Unit production Removal.

1) what is unit production?

- A production in the form:

Non terminal → single non terminal

is called an unit production.

Example, $B \rightarrow C$

$C \rightarrow A.$

2) what are the problems with unit productions?

- Unit production increases the number of steps as well as complexity at the time of generating language from the grammar.

3) what is the procedure to remove an unit production from a given CFG.

- Algorithm

```
while (there exist a unit production NT → NT)
{
    select a unit production A → B
    for (every non-unit production B → α)
    {
        add production A → α to the grammar
        eliminate A → B from the grammar
    }
}
```

problems

1) Remove unit production from the grammar.

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

step 1: In this CFG. the unit productions are,

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

step 2: In the first iteration of the while loop, we select the unit production $A \rightarrow E$

Now we have to find whether there exists a production as follows in the above CFG.

$E \rightarrow$ string of terminal/non-terminal.

$\exists$ a production of the above form

$E \rightarrow a$.

Hence the new production rule will be generated and added to the grammar, i.e. $A \rightarrow a$.

we shall delete $A \rightarrow E$

step 3: In the second iteration of the while loop, we select the unit production $B \rightarrow C$

Now we have to find whether $\exists$ a production as follows in the above CFG.

$C \rightarrow$ string of terminals/non terminals and terminals

No such production rule is found

So we shall eliminate $B \rightarrow C$ later on.

Step 4: In the third iteration of the while loop, we select the unit production C→D

Now we need to find whether ∃ a production as follows in the above CFG

D → string of terminals/terminal & non terminal.

D→b is an example of that.

So the new production rule can be generated as C → b.

we will eliminate C→D

Step 5: Now B→C can be deleted as C→b is generated and added to the CFG.

∴ we delete B→C

and we add B→b.

step 6: After deleting all the unit production 1 by 1, the modified production rule of the CFG will be

S → AB

A → a.

B → b.

C → b.

D → b

E → a.

Step 7: This grammar does not have any unit production but it is not normalised as there are non reachable symbols C, D, E

∴ we have to eliminate them

So the final CFG after removing the non reachable symbols are

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

2) Remove unit production from the following grammar:

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

Step 1: In this CFG the unit productions are,

$B \rightarrow C$
$C \rightarrow D$

Step 2: Now in the first iteration we cannot do anything so we leave

$B \rightarrow C$.

Step 3: Now in the second iteration we remove $C \rightarrow D$

we get $C \rightarrow b$.

Step 4: Thus $B \rightarrow b$ is added.

Step 5: $C \rightarrow b$ is not reachables from S.

Step 6: Final grammar is

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

3) Remove unit production from the following grammar,

~~S → AA~~

~~A → BB/Ab/b~~

S → aX/Yb/Y

X → S

Y → Yb/b

S → Y

X → S

↓

S → b.

X → b.

∴ S → aX/Yb/b.

X → aX/Yb/b. [∵ X → S]

Y → Yb/b.

Removal of Null production.

1) What do you mean by Null production?
A production in the form:
Non-terminal → Null (ε)
is called Null production.

2) When Null production cannot be removed?
-i) If Null string (ϕ) is in the language set generated from the grammar, then Null production cannot be removed.
ii) If we get $S \overset{*}{\rightarrow} \varepsilon$, then the Null production cannot be removed

3) What is the procedure to remove Null production?
Step 1: If $A \rightarrow \varepsilon$ is a production to be eliminated, then we look for all productions whose RHS contains A
Step 2: Replace each occurance of A in each of these productions to obtain the non null production.
Step 3: These non null productions must be added to the grammar to keep the language generating power same.

Example

1) Remove null production from the following grammar:

$$S \rightarrow aA$$
$$A \rightarrow b / \varepsilon$$

Ans
Step 1: In the production rule of the grammar, the production rule with null production is $A \rightarrow \varepsilon$

Step 2: The grammar doesnot produce Null string as a language. so this null production can be removed.

Step 3: According to the step1 of this algorithm we need to look for all the Productions whose RHS contains A. There exists only 1 production like that in the above CFG, which is $S \to aA$

Step 4: According to step 2 of the algorithm we need to replace each occurance of A in each of the productions. There is only 1 occurance of A in $S \to aA$ So after replacing A with ϵ, in this production rule it becomes

$$S \to a$$

Step 5: According to step 3 of algorithm $S \to a$ will be added to the CFG and $A \to \epsilon$ will be deleted from the CFG.

so the new production will be

$S \to aA / a$

$A \to b$

2) Remove null production from the following grammar.

$S \to aX / bX$

$X \to a / b / \epsilon$

$X \to \epsilon$

$\therefore S \to aX / a$

$S \to bX / b$

$\therefore S \to a/b/ax/bx$

$X \to a/b.$

Normal Form

1) What do you mean by Normal form of a grammar?

- For a grammar, the RHS of a production can be any string of variables & terminals i.e. to say $(V_n \cup \Sigma)^*$

A grammar is said to be in normal form when every production of the grammar has some specific form, this means except allowing any number of $(V_n \cup \Sigma)$ on the RHS of the production we permit only specific numbers on the RHS of the production.

But these restrictions should not hamper the language generating power of the grammar.

*2) Define Chomsky Normal form.

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

i) Non-terminal → string of exactly 2 non-terminals (A→BC)

ii) Non-terminal → single terminal. (A→a)

3) How to convert a CFG into a CNF?

Step 1: Eliminate all null production
Eliminate all unit production
Eliminate all useless symbols

step 2: If all productions are in the form,

i) NT → string of Exactly 2 NTs

or

ii) NT → single Terminal

Then declare the CFG is already in CNF and stop the algorithm else follow step 3. and/or step 4 and/or step 5

step 3: Elimination of terminals on the RHS of length 2 or more

step 4: Restriction of no. of variables on the RHS to 2.

step 5: Convession of string containing terminals and nonterminals to string of non-terminals on the RHS

Problem: 1

Convert the following grammar into CNF.

$S \rightarrow bA/aB$

$A \rightarrow bAA/aS/a$

$B \rightarrow aBB/bS/a$

step 1: In the above CFG there is no null null production, no unit productions or no useless symbols.

step 2: The productions

$S \rightarrow bA$ ✓ $B \rightarrow aBB$ ✓

$S \rightarrow aB$ ✓ $B \rightarrow bS$.

$A \rightarrow bAA$ ✓

$A \rightarrow aS$ ✓

are not in the form of CNF.

so we have to convert these into CNF.

Step 3: Let us replace terminal a by non terminal C_a

Let us replace the terminal b by non terminal C_b

Hence 2 new production rules will be added to the grammar.

$C_a \rightarrow a$

$C_b \rightarrow b$.

By replacing a & b by new nonterminals and including 2 new productions in the grammar, the modified grammar will be.,

$S \rightarrow C_bA / C_aB$

$A \rightarrow C_bAA / C_aS / a$

$B \rightarrow C_aBB / C_bS / a$

$C_a \rightarrow a$

$C_b \rightarrow b$

Step 4: In the modified grammar all the productions are not in CNF, for example

$A \rightarrow C_bAA$

$B \rightarrow C_aBB$

are not in CNF

Because they contain more than 2 nonterminals at the RHS.

Lets replace AA by a new non-terminal D. so we add a new rule,

$D \rightarrow AA$

Replace BB by new non terminal E.

so we add a rule

$E \rightarrow BB$

∴ The modified grammar will be

$S \to C_bA / C_aB$

$A \to C_bD / C_aS / a$

$B \to C_aE / C_bS / a$

$C_a \to a$

$C_b \to b$

$D \to AA$

$E \to BB$

2) Convert the following grammar into CNF

$S \to ABa$

$A \to aab$

$B \to Ac$

$C \to a$

There is no null production
no unit symbol
no useless symbol

Let $D \to b$.

Let $E \to AB$

$S \to ABa$

Let $F \to CC$

$\to ABC$ [∵ $C \to a$]

$\to EC$ [∵ $E \to AB$]

$A \to aab$

$\to CCD$ [$C \to a$, $D \to b$]

$\to FD$ [$F \to CC$]

∴ Final grammar

$S \to EC$ $E \to AB$

$A \to FD$ $F \to CC$

$B \to AC$

$C \to a$

$D \to b$

(i) $S \to ABa$ is not in CNF

So, we assume; $D_a \to a$

$S \to AB D_a$

(ii) ⑧ $A \to a a b$ is not in CNF

So, we assume $D_b \to b$

⑧ $A \to D_a D_a D_b$

So, CFG will be,

$S \to AB D_a$

$A \to D_a D_a D_b$

$B \to AC$

$C \to a$

$D_a \to a$

$D_b \to b$

In the above grammar
$S \to ABD_a$ is not in CNF.

We assume, $E \to AB$.

$S \to EDa$

$E \to AB$

In the above grammar
$A \to D_aD_aD_b$ is not in CNF

We assume, $F \to D_aD_a$

$A \to FD_b$

$F \to D_aD_a$

Now, CFG is,

$S \to EDa$

$E \to AB$

$A \to FD_b$

$F \to D_aD_a$

$B \to AC$

$D_a \to a$

$D_b \to b$

$C \to a$

Convert the given CFG into CNF.

E → E+E
E → E*E
E → id

P → +
M → *

∴ E → E+E
E → EPE
E → AE

E → E*E
E → EME
E → BE

A → EP
B → EM.

∴ E → AE
E → BE
E → id.
P → +
M → *
A → EP
B → EM.

Convert the given grammar into CNF.

S → abAB
A → bAB/ε
B → BAa/ε

There are 2 null productions

A → ε
B → ε.

Putting them in RHS and removing them.

S → abAB/abB/abA/ab.
A → bAB/bB/bA/b.
B → BAa/Ba/Aa/a.

$C_a \to a$ $\quad D \to C_a C_b$

$C_b \to b$ $\quad E \to AB$ $\quad F \to BA$

$S \to C_a C_b AB$ $\quad S \to DE$ ✓

$S \to C_a C_b A$ $\quad S \to DA$ ✓

$S \to C_a C_b B$ $\quad S \to DB$ ✓

$S \to C_a C_b$ ✓

$A \to C_b AB$ $\quad A \to C_b E$ ✓

$A \to C_b A$ ✓

$A \to C_b B$ ✓

$A \to b$ ✓

$B \to BAC_a$ $\quad B \to FC_a$

$B \to BC_a$ ✓

$B \to AC_a$ ✓

$B \to a$ ✓

∴ $S \to DE / DA / DB / C_a C_b$

$A \to C_b E / C_b A / C_b B / b$

$B \to FC_a / BC_a / AC_a / a$

$C_a \to a$

$C_b \to b$

$D \to C_a C_b$

$E \to AB$

$F \to BA$

Greibach Normal form.

Definition (GNF): A grammar is said to be a GNF if every production(s) of the grammar of the form, i) NT → (single T)(string of NT)
ii) NT → single T.

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

NT → (single Terminal)(NT)*

Note: where closure (*) indicates any combination of non terminals including null.

What is the application of GNF?

If a CFG is converted to GNF, then CFL which is generated from the GNF is easily accepted by PDA

Describe the Lemmas in order to convert a CFG into GNF

Lemma I: If $G = (N, T, P, S)$ is a CFG and NF $A \to A\alpha$ (A → NT, α → string of NT) and $A \to \beta_1 / \beta_2 / \beta_3 \ldots \beta_n$ belongs to the production rules (P) of G, then a new grammar $G' = (N, T, P', S)$ can be constructed by replacing $A \to \beta_1, \beta_2 \ldots \beta_n$ in $A \to A\alpha$, which will produce, $A \to \beta_1\alpha / \beta_2\alpha / \ldots \beta_n\alpha$. [β → string of NT]

Lemma II: Let $G = (N, T, P, S)$ be a CFG and the set of A productions which belongs to P be

$A \to A\alpha_1 / A\alpha_2 / A\alpha_3 / \ldots A\alpha_n$ or $\beta_1 / \beta_2 / \ldots \beta_n$.

then we introduce a new nonterminal X.

Let $G' = (N', T, P', S)$, where $N' = (N \cup X)$

P' can be formed by replacing the A

Productions by $A \to B_i$ $\quad X \to \alpha_j$

$A \to B_i X$ $\quad X \to \alpha_j X$

What is the process of converting a given CFG into GNF?

Step 1: Eliminate all Null productions, Unit productions and useless symbols from the given grammar and convert the grammar into CNF.

Step 2: Rename non-terminals of N as $A_1, A_2, A_3, \ldots A_n$ with start symbol, A_1.

Step 3: Using Lemma 1 modify the productions such that LHS variables subscript is less than the RHS starting variable subscript i.e. to say, all the productions will be in the form, $A_i \to A_j V$ where $i \leq j$

Step 4: By repeating application of Lemma 1 and Lemma 2, all the productions of the modified grammar will come into GNF.

Problem 1

Convert the given grammar into GNF.

$S \to AA/a$

$A \to SS/b$

Step 1: In these production rules stated above

∃ no unit productions like $A \to B$

∃ no production like $A \to \epsilon$, so there is no null production.

∃ no non reachable and non generating symbols as in the above production both A & S are generating a terminal b & a respectively, so there is no non generating symbol.

S is reachable from A and vice versa. So there is no non reachable symbol.

Hence the above grammar G is free from unit production, useless symbols and NULL productions.

A grammar is said to be in CNF if it is in the following form, $NT \rightarrow NT_1\ NT_2$ or $NT \rightarrow T$

The given grammar's production rules $S \rightarrow AA$ satisfies rule 1 of CNF so it is already in CNF. $S \rightarrow a$ satisfies rule 2 of CNF so it is already in CNF. $A \rightarrow SS$ satisfies rule 1 of CNF so it is already in CNF. $A \rightarrow b$ satisfies rule 2 of CNF, so it is already in CNF

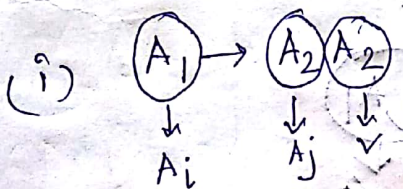
Step 2: According to step 2 we found that there are two non terminals in the production rule, which are A, S. Here starting symbol is S which is renames as A_1 and A is renamed as A_2

After renaming the non terminals of the old production rules, new production rules will be

$A_1 \rightarrow A_2 A_2 / a$

$A_2 \rightarrow A_1 A_1 / b$.

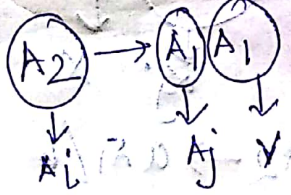
Step 3: step 3 is completed when all the productions are in the form $A_i \rightarrow A_j V$ where $(i \leq j)$.

(i) 

$i = 1$ $A_i \leq j$

$j = 2$

(ii) $A_1 \rightarrow a$ (Already in GNF)

(iii) 

Where $i = 2$

$j = 1$

$i \geq j$

So, $A_2 \rightarrow A_1 A_1$ is not in the form of $A_i \rightarrow A_j V$

(iv) $A_2 \rightarrow b$ (Already in GNF)

Replace the leftmost A_1 at the RHS of the production $A_2 \rightarrow A_1 A_1$.

After replacing the modified A_2 productions will be,

$A_2 \rightarrow \underline{A_1} A_1$

$\rightarrow A_2 A_2 A_1$

$A_2 \rightarrow \underline{A_1} A_1$

$\rightarrow a A_1$

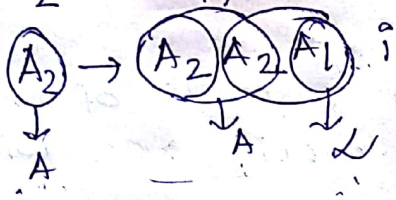
So, $A_2 \rightarrow A_1 A_2 A_1 / a A_1 / b$

$A_1 \rightarrow A_2 A_2 / a$

Step 4:

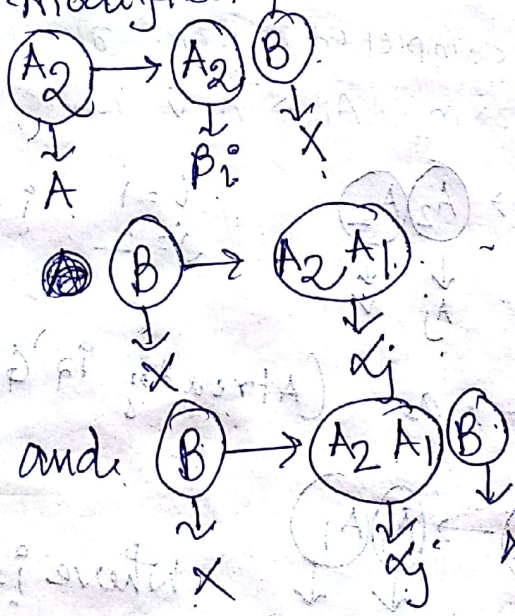
(i) $A_2 \rightarrow a A_1 / b$ is in GNF ✓

(ii) $A_2 \rightarrow A_2 A_2 A_1$ is in the format $A \rightarrow A\alpha$



So, we introduce a new NT B.

So, the modified production rule will be



by replacing $A_2 \rightarrow aA_1 / b$ in the production $A_2 \rightarrow A_2 B$ we get

$A_2 \rightarrow a A_1 B$ ✓

$A_2 \rightarrow bB$ ✓

So, $A_2 \rightarrow aA_1/b/aA_1B/bB$ is in GNF.

(iii) By replacing $A_2 \rightarrow aA_1/b/aA_1B/bB$ in the production rule $A_1 \rightarrow A_2A_2/a$ we get,

a) $A_1 \rightarrow \underline{A_2}A_2$
$\rightarrow aA_1A_2$

b) $A_1 \rightarrow \underline{A_2}A_2$
$\rightarrow bA_2$

c) $A_1 \rightarrow \underline{A_2}A_2$
$\rightarrow aA_1BA_2$

d) $A_1 \rightarrow \underline{A_2}A_2$
$\rightarrow bBA_2$

e) $A_1 \rightarrow a$

So, $A_1 \rightarrow aA_1A_2/bA_2/aA_1BA_2/bBA_2/a$ is in GNF

(iv) By replacing $A_2 \rightarrow aA_1/b/aA_1B/bB$ in the production rule $B \rightarrow A_2A_1/A_2A_1B$. we get,

a) $B \rightarrow \underline{A_2}A_1$
$\rightarrow aA_1A_1$

b) $B \rightarrow \underline{A_2}A_1$
$\rightarrow bA_1$

c) $B \rightarrow \underline{A_2}A_1$
$\rightarrow aA_1BA_1$

d) $B \rightarrow \underline{A_2}A_1$
$\rightarrow bBA_1$

e) $B \rightarrow \underline{A_2}A_1B$
$\rightarrow aA_1A_1B$

f) $B \rightarrow \underline{A_2}A_1B$
$\rightarrow bA_1B$.

g) $B \rightarrow \underline{A_2}A_1B$
$\rightarrow aA_1BA_1B$

h) $B \rightarrow \underline{A_2}A_1B$
$\rightarrow bBA_1B$.

So, $B \rightarrow aA_1A_1 \mid bA_1 \mid aA_1BA_1 \mid bBA_1$

$B \rightarrow aA_1A_1B \mid bA_1B \mid aA_1BA_1B \mid bBA_1B$

Are in GNF.

Closure properties of CFL

Prove that CFLs are closed under union, concatenation and star-closure.

union:

Let L_1 is a CFL for the CFG $G_1 = (V_{N_1}, \Sigma_1, P_1, S_1)$ and L_2 is a CFL for the CFG $G_2 = (V_{N_2}, \Sigma_2, P_2, S_2)$ we have to prove that $L = L_1 \cup L_2$ is also in CFL.

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

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

$\Sigma = \Sigma_1 \cup \Sigma_2$

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

From the above discussion it is clear that the Language set generated from the grammar G contains all the strings that are derived from S_1 as well as S_2. Hence it is proved that $L = L_1 \cup L_2$.

Concatenation:

Let L_1 is a CFL for the CFG $G_1 = (V_{N_1}, \Sigma_1, P_1, S_1)$ and L_2 is a CFL for the CFG $G_2 = (V_{N_2}, \Sigma_2, P_2, S_2)$ we have to prove that $L = L_1L_2$ 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,

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

$\Sigma = \Sigma_1 \cup \Sigma_2$,

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

From the above discussion it is clear that the language set generated from the grammar G_1 contains all the strings that are derived from S_1 as well as S_2. Hence it is proved that $L = L_1 L_2$.

Star-closure:

Let L is a CFL for the CFG $G_1 = (V_{N_1}, \Sigma_1, P_1, S_1)$. we have to prove that L^* is also in CFL.

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

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

$P = \{S \to S_1 S / \varepsilon\} \cup P_1$.

Obviously, G_1 is a CFG. It is easy to see that $L(G_1) = L^*$

Prove that CFL are not closed under intersection and complementation.

Intersection:

Consider two languages

$L_1 = \{a^n b^m c^n$, where $n, m \geq 0\}$ and

$L_2 = \{a^n b^n c^m$, where $n, m \geq 0\}$.

we can easily show that the two languages L_1 and L_2 are context free. (By constructing grammar)

Consider $L = L_1 \cap L_2$.

Hence $L = a^n b^m c^n \cap a^n b^n c^m$. where $n \geq 0$.

The $a^n b^n c^n$ is a context sensitive language not a context free. From here we can conclude that CFLs are not closed under intersection.

Complementation:

From set theory we can prove $L_1 \cap L_2 = \bar{L}_1 \cup \bar{L}_2$ by (D' Morgan's Law).

If the union of the complements of L_1 & L_2 are closed, i.e. also context free, then the LHS will also be context free. But we have proved that $L_1 \cap L_2$ is not context free. we are getting contradiction here. Hence CFLs are not closed under complementation

Prove that every regular language is a CFL.

From recursive definition of Regular set we know ϕ and $\in$ are regular expression. If R is regular expression, then R+R (union), R.R (concatenation) and R* (Kleene star) are also regular expressions. A regular expression R is a string of terminal symbols. The ϕ and $\in$ are also CFL, and we know that CFL are closed under union, concatenation, and Kleene stare. ∴ Every regular language is a CFL.

State OGDEN's Lemma for CFL.

Ogden's lemma states that if a language L is context free, then ∃ some number p>0 (where p may or may not be a pumping length) such that for any string w of length atleast p in L and every way of making p or more of the positions in w, w can be written as, uvxyz.

where u, v, x, y & z are strings such that,

i) x has atleast 1 marked position

ii) Either u and v both have marked positions or y & z both have marked positions.

iii) vxy has atmost p marked positions.

iv) $uv^i xy^i z$ is in L for every $i \geq 0$.

Q: Simplify the following CFG

$S \rightarrow AaB/aaB$

$A \rightarrow D$

$B \rightarrow bbA/\varepsilon$

$D \rightarrow E$

$E \rightarrow F$

$F \rightarrow aS$

Q: Convert the following grammar into CNF.

$S \rightarrow aA/bB$

$A \rightarrow aBB/bS/b$

$B \rightarrow bAA/aS/a$

Q: Convert the following grammar into CNF.

$S \to aA/B/C/a$
$A \to aB/E$
$B \to aA$
$C \to cCD$
$D \to abd.$

Q: Convert the following grammar into CNF.

$E \to E+T/T$
$T \to (E)/a$

Q: Convert the following grammar into CNF.

$S \to ABb/a$
$A \to aaA/B$
$B \to bAb/b$

Q: Convert the following grammar into CNF.

(i)
$S \to A$
$A \to aBa/\wedge$
$B \to bAb/b$

(ii)
$S \to XY$
$X \to YS/b$
$Y \to SX/a$

(iii)
$S \to AB/BC$
$B \to bB/cC/b$
$A \to aB/bA/a$
$C \to c$

Q: Remove left recursion from the given grammar.

(i) $A \rightarrow Ba / b$
$B \rightarrow Bc / Ad / e$

(ii) $E \rightarrow E + T / T$
$T \rightarrow T * F / F$
$F \rightarrow id / (E)$

(iii) $S \rightarrow Aa / b$
$A \rightarrow Sc / d$

Push Down Automata

Module-3 Ch-2

Objectives

1) Definition of PDA
2) Acceptance of CFL.
3) Acceptance by Final state and Acceptance by empty state and its equivalence
4) Equivalence of CFL & PDA.
5) Introduction to DCFL & DPDA

Definition of PDA

i) PDA is a machine format for CFL.
ii) A CFL is generated from CFG.
iii) A CFL is accepted by a logical machine PDA.
iv) PDA can be of 2 types,
   a) Deterministic PDA (DPDA)
   b) Non Deterministic PDA (NDPDA)

[For every possible CFL it is not possible to generate DPDA but it is possible to generate NDPDA.

For every RL it is possible to generate either NDFA or DFA]

[No. of states in NDPDA will always be greater than no. of states in DPDA]

DPDA -> n no. of states
NDPDA -> 2^n " " "
then $n < 2^n$.

DFA -> n no of states
NDFA -> 2^n " " "
then $n \leq 2^n$

- Describe the block diagram or mechanical diagram of PDA.

1) Input Tape → i) Contains the input symbols. (a a a b b b)
ii) The tape is divided into no. of equal sized squares
iii) Each square contains a single i/p character.
iv) String placed in the i/p tape is traversed from left to right.
v) 2 end side of the i/p string contain infinite no. of blank symbols.

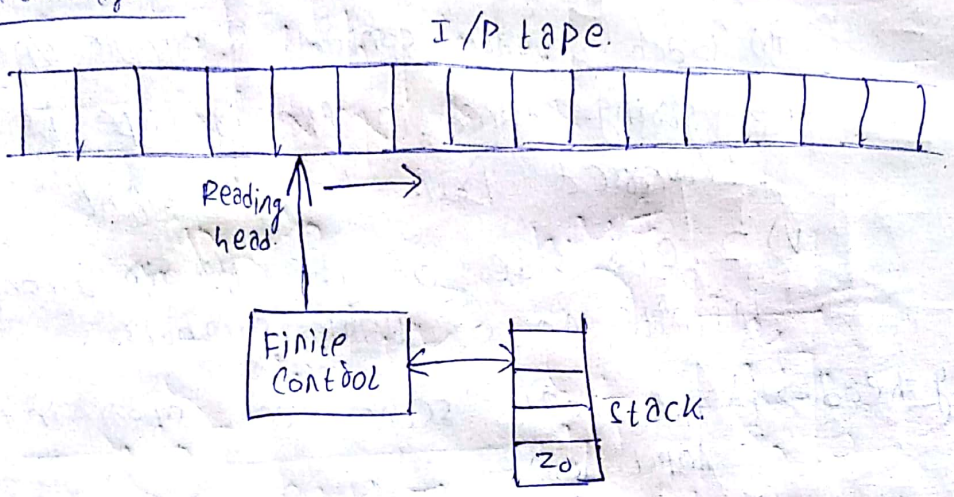
2) Reading head → i) The head scans each square in the input tape.
ii) It reads the i/p from the tape.
iii) The head moves from left to right
iv) I/P scanned by the reading head is sent to the Finite Control unit of PDA.

3) Finite control unit → i) Finite control can be considered as a control unit of a PDA
ii) It is an automaton always reside in a state.
iii) Reading head scans the i/p from the i/p tape and sends it to Finite control unit.
iv) The input sent by reading head to the finite control unit from the i/p tape is pushed into the stack TOP.
v) Depending on the input taken from the i/p tape and i/p from the stack top the finite control unit decides that in which state PDA will move, and which input symbol it will push to the stack or Pop from the stack or do nothing on the stack

4) Stack → i) A stack is a temporary storage of stack symbols.
ii) Every move of PDA indicates on of the following operation to the stack

a) push → 1 stack symbol may be added to the stack
b) POP → 1 stack symbol may be deleted from the top of stack
c) skip → 1 input symbol may be skipped or we may do nothing on the stack.

Block Diagram



- State the mathematical equation of PDA

A PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a seven-tuple structure where,

1) Q is the finite set of states
2) Σ is the character set of the language for which the PDA is designed.
3) Γ is the symbols that are pushed down into the stack. (Finite)
4) δ is known as the set of transition functions. It governs the working of the PDA
5) q_0 is the starting state of the read head
6) Z_0 is known as the initial pushdown symbol.
7) F is known as the set of final states.

Acceptance by a PDA

There are 2 ways to declare a string accepted by a PDA.

A) Accepted by empty stack / empty store.

B) Accepted by Final state.

A) Accepted by empty stack/empty store.

we know in each PDA there is a stack, in the stack at the bottom there is a symbol called stack bottom symbol. In each move of the PDA one symbol called stack symbol is either pushed in or popped from the stack. But the symbol z_0 still remains in the stack.

A string w may be declared accepted by the empty stack after processing all the symbols of w. If the stack is empty after reading the rightmost input character of the string w.

Mathematically we can define it as, the string w is declared accepted by empty stack if:

$$\{w \in \Sigma^* / (q_0, w, z_0)^* \rightarrow (q, \lambda, \lambda) \text{ for some } q \in Q\}$$

In general we can say that a string is accepted by a PDA of empty stack if both the following conditions are fulfilled.

i) The string is finished (Totally traversed)

ii) stack is empty.

b) Accepted by final state. A string w may be declared accepted by final state if after total traversal of the string w, the PDA enters into its final state. In mathematical notation, we can say $M = \{Q, \Sigma, \Gamma, \delta, q_0, z_0, F\}$ be a PDA, the string w is declared accepted by final state if,

$\{ w \in \Sigma^* / (q_0, w, z_0)^* \rightarrow (q_F, \lambda, z_0)$ for $q_F \in F$ and z_0 is the stack bottom symbol$\}$

In general we can say that a string is accepted by a PDA by final state if both the following conditions are fulfilled.

i) The string is finished (totally traversed)

ii) The PDA enters into its final state.

(Though there are 2 ways to declare a string accepted by a PDA, but it can be proved that both the ways are equivalent. In general for both the cases, all the steps are same, except the last step. In the last step, it is differentiated whether the string is accepted by empty stack or by final state.

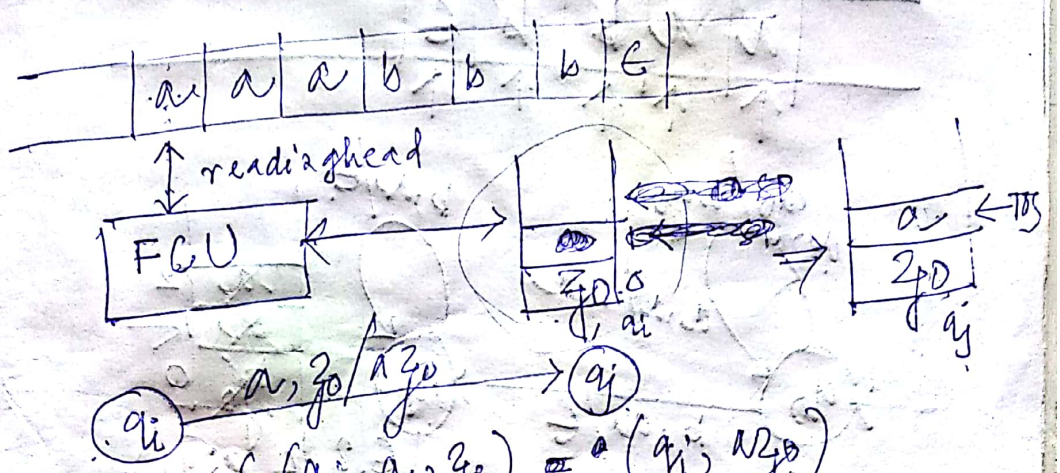
if in the last step

$\delta(q_i, \lambda, z_0) \rightarrow (q_i, \lambda)$, it is accepted by empty stack.

if $\delta(q_i, \lambda, z_0) \rightarrow (q_f, z_0)$, it is accepted by final state)

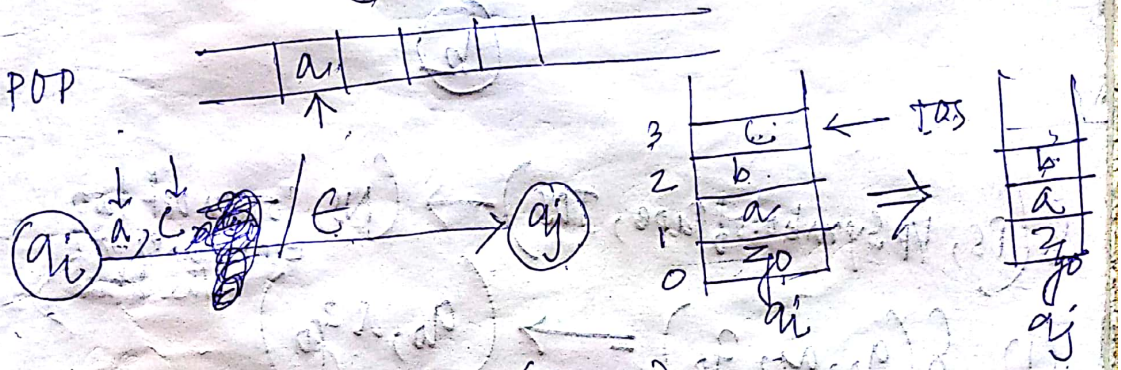
Stack related operations on PDA

PUSH



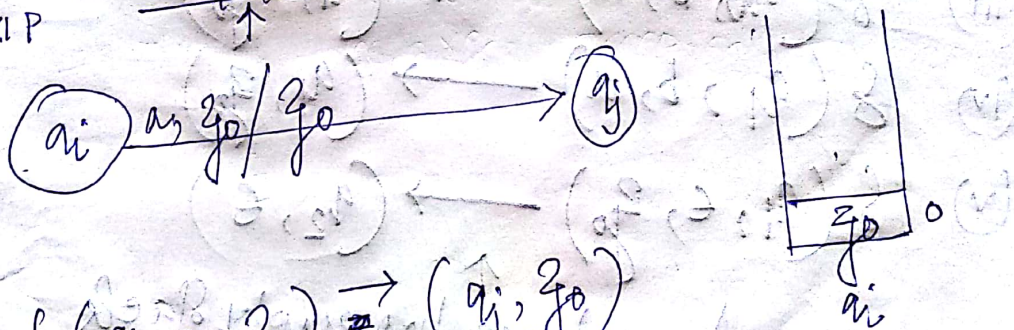
$$\delta(q_i, a, z_0) \rightarrow (q_j, a z_0)$$

POP

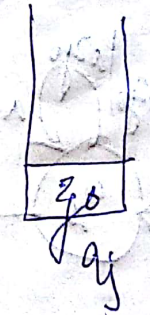


$$\delta(q_i, a, c) \rightarrow (q_j, \epsilon)$$

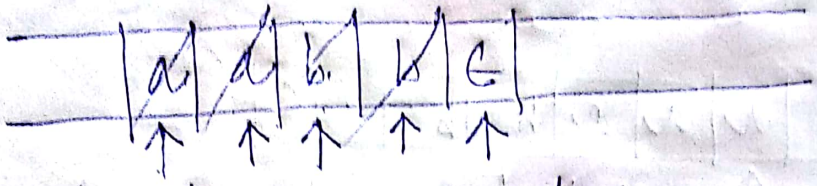
SKIP

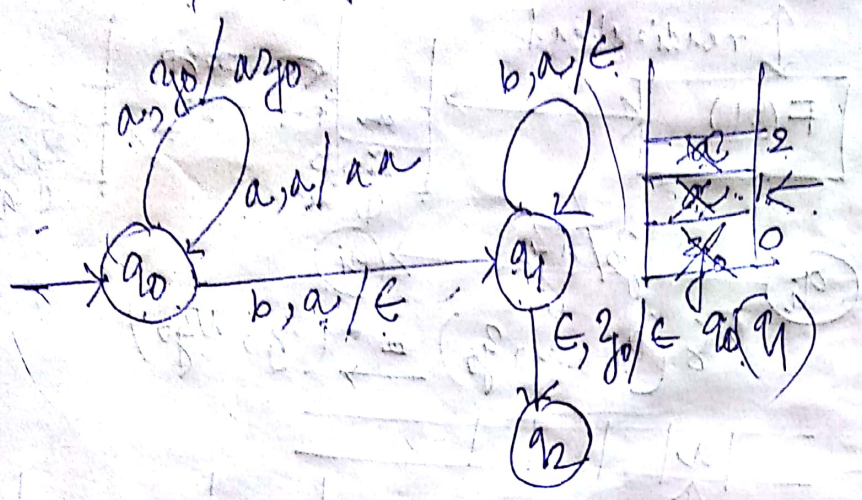


$$\delta(q_i, a, z_0) \rightarrow (q_j, z_0)$$



Construct a PDA of a given language by empty stack and final state, both where $L = a^n b^n$ $n \geq 1$





δ(PS, I/P Symbol of Tape, TOS) → (NS, Top most two elements of the Stack)

(i) $\delta(q_0, a, z_0) \longrightarrow (q_0, a z_0)$

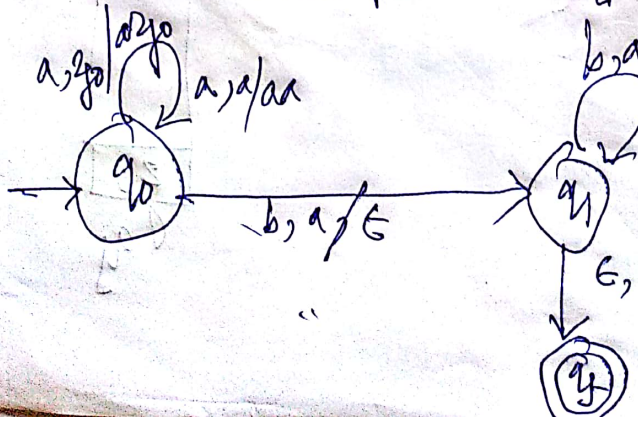
(ii) $\delta(q_0, a, a) \longrightarrow (q_0, aa)$

(iii) $\delta(q_0, b, a) \longrightarrow (q_1, a z_0)$

(iv) $\delta(q_1, b, a) \longrightarrow (q_1, z_0)$

(v) $\delta(q_1, \epsilon, z_0) \longrightarrow (q_2, \epsilon)$

Acceptance by empty Stack.



(i) $\delta(q_0, a, Z_0) \rightarrow (q_0, aZ_0)$

(ii) $\delta(q_0, a, a) \rightarrow (q_0, aa)$

(iii) $\delta(q_0, b, a) \rightarrow (q_1, aZ_0)$

(iv) $\delta(q_1, b, a) \rightarrow (q_1, Z_0)$

(v) $\delta(q_1, \epsilon, Z_0) \rightarrow (q_2, Z_0)$

Acceptance by final state

DPDA

$\delta : Q \times \Sigma \times \Gamma \Rightarrow Q \times \Gamma^*$

NPDA

$\delta : Q \times \Sigma \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$

Mishra

pg-248 (*) 244 241 237 234 (*) 233 (*) 260 261 262 263

7.9 7.8 7.7 7.5 7.4 7.3 7.15 7.16 7.17 7.18

Nagpal

198* 199* 200 201 202 207 209

6.1 6.2 6.3 6.4, 6.5 6.6, 6.7 6.10 (DCFL, DPDA Read.)

222

6.16, 6.17, 6.18