Context Free Grammars and Languages

Grammars are used to generate the words of a language and to determine whether a word is in a language. Formal languages, which are generated by grammars, provide models for both natural languages, such as, English, and for programming languages, such as C, JAVA.
Context free grammars are used to define syntax of almost all programming languages, in context of computer science. Thus an important application of context free grammars occurs in the specification and compilation of programming languages.

Formal Description of Context Free Grammar:

A context free grammar is defined by 4-tuples (V, T, P and S) where,

V= set of variables
T= set of terminal symbols
P= set of rules and productions
S= start symbol and S ε V

Thus, CFG consists of a collection of substitution rules, also called productions, with each rule being a variable and a string of variables and terminals. Each rule appears as a line in the grammar.
The symbols involved in the productions may be variable or terminal symbols. The variable symbols are after represented by capital letters. The terminals are analogous to the input alphabet and are often represented by lower case letters. One of the variables is designed as a start variable. It usually occurs on the left hand side of the topmost rule.

For example,

$S \rightarrow \epsilon$
$S \rightarrow 0S1$

This is a CFG defining the grammar of all the strings with equal no of 0's followed by equal no of 1's.
Here, the two rules define the production P,
ϵ, 0, 1 are the terminals defining T,
S is a variable symbol defining V
And S is start symbol from where production starts.

The CFG are more powerful than the regular expressions as they have more expressive power than the regular expression. Generally regular expressions are useful for describing the structure of lexical constructs as identical keywords, constants etc. But they do not have the capability to specify the recursive structure of the programming constructs.
However, the CFG are capable to define any of the recursive structure also. Thus, CFG can define the languages that are regular as well as those languages that are not regular.

Example

CFG representing the language over $\Sigma = \{a, b\}$ which is palindrome language.

$S \rightarrow \epsilon \mid a \mid b$
$S \rightarrow a S a$

S➔ b S b

Meaning of context free:

Consider an example:

S ➔ a M b

M ➔ A | B

A➔ Є | aA

B➔ Є | bB

How, consider a String aaAb, which is an intermediate stage in the generatin of aaab. It is natural to call the strings "aa" and "b" that surround the symbol A, the "context" of A in this particular string. Now, the rule A➔ aA says that we can replace A by he string aA no matter what the surrounding strings are; in other words, independently of the context of A.

When there is a production of form lw_1r➔ lw_2r (but not of the form w_1➔ w_2), the grammar is context sensitive since w_1 can be replaced by w_2 only when it is surrounded by the strings "l" and"r".

Derivation Using Grammar Rule:

General Convention

- Capital letters near the beginning of the alphabet A, B and so on are used a variables.
- Capital letters near the end of the alphabet such as X, Y are used as either terminals or variables.
- Lower case letters near the beginning of the alphabet, a, b and so on are used fr terminals.
- Lower case letters near the end of alphabet, such as 'w' or 'z' are used for string of terminals.
- Lower case Greek letters such as α and β are used for string of terminal or variables.

Derivation:

A derivation of a context free grammar is a finite sequence of strings $\beta_0\ \beta_1\ \beta_2 \ldots\ldots\ldots \beta_n$ such that:

- For $0 \le i \le n$, the string $\beta_i\ \varepsilon\ (V\ U\ T)^*$
- $B_0 = S$
- For $0 \le i \le n$, there is a production of P that applied to β_i yields β_{i+1}
- $B_n\ \varepsilon\ T^*$

There are two possible approaches of derivation:

- Body to head (Bottom Up) approach.
- Head to body (Top Down) approach.

Body to head

Here, we take strings known to be in the language of each of the variables of the body, concatenate them, in the proper order, with any terminals appearing in the body, and infer that the resulting string is the language of the variables in the head.

Consider grammar,

S➔ S + S; S➔ S/S; S➔ (S)

S➔S-S; S➔ S*S; S➔ a

Here given a + (a*a) / a – a

Now, by this approach.

S.N	String Inferred	Variable	Production	String Used
1	a	S	S➔ a	a; string 1
2	a*a	S	S➔S*S	a*a; string 2
3	(a*a)	S	S➔(S)	String 1 and string 2; string 3
4	(a*a) / a	S	S➔S/S	String 1 and string 3; string 4
5	a + (a*a) / a	S	S➔S+S	String 1 and String 4; string 5
6	a + (a*a) / a - a	S	S➔S-S	String 5 and string 1

Thus, in this process we start with any terminal appearing in the body and use the available rules from body to head.

Head to Body

Here, we use production from head to body. We expand the start symbol using a production, whose head is the start symbol. Here we expand the resulting string until all strings of terminal are obtained. Here we have two approaches

Leftmost derivation

Rightmost derivation

Leftmost Derivation: Here leftmost symbol (variable) is replaced first.

Rightmost Derivation: Here rightmost symbol is replaced first.

For example

#Given the grammar:

S ➔ S*S
S➔ S+S
S➔ S-S
S➔ S/S
S➔ (S)
S➔ a

Now, we perform head to body derivation for

a + (a*a) / a-a

Now leftmost derivation for the given string is:

S ➔ S + S; rule S➔ S + S
S➔ a + S; rule S➔ a
S➔ a + S - S; rule S➔ S - S
S➔ a + S / S - S; rule S➔ S / S
S➔ a + (S) /S – S; rule S➔ (S)
S➔ (S*S) / S – S; rule S➔ S*S
S➔ a + (a*S) / S – S; rule S➔ a
S➔ a + (a*a) / S - S;
S➔ a + (a*a) / a – S;
S➔ a + (a*a) / a – a;

And rightmost derivation is;

S➔ S – S; rule S➔ S - S
S➔ S – a; rule S➔ a
S➔ S + S – a; rule S➔ S + S
S➔ S + S / S – a; rule S➔ S / S
S➔ S + S / a –a; rule S➔ a
S➔ S + (S) / a – a; rule S➔ (S)
S➔ S + (S*S) / a – a; rule S➔ S*S
S➔ S + (S*a) / a – a; rule S➔ a
S➔ S + (a*a) / a – a;
S➔ a + (a*a) / a – a;

#Consider a Grammar;

S➔ aAS | a
S➔ SbA | SS | ba

Given a string aaabaaa, give leftmost and rightmost derivation.

Leftmost Derivation:

$S \Rightarrow_{lm}$ aAS
$S \Rightarrow_{lm}$ aSSS; rule A➔ SS
$S \Rightarrow_{lm}$ aaSS; rule S ➔ a
$S \Rightarrow_{lm}$ aaaSS; rule S➔ aAS
$S \Rightarrow_{lm}$ aabaSS; rule A➔ ba
$S \Rightarrow_{lm}$ aaabaaS; rule S➔ a
$S \Rightarrow_{lm}$ aaabaaa; rule S➔ a

Rightmost Derivation:

$S \Rightarrow_{rm}$ aAS
$S \Rightarrow_{rm}$ aAa; rule S ➔ a
$S \Rightarrow_{rm}$ aSSa; rule A➔ SS
$S \Rightarrow_{rm}$ aSaASa; rule S➔ aAS
$S \Rightarrow_{rm}$ aSaAaa; rule S➔ a
$S \Rightarrow_{rm}$ aSabaaa; rule A➔ ba
$S \Rightarrow_{rm}$ aaabaaa; rule S ➔a

 # Given CGF:

S➔ aB | bA
A➔ a | aS | bAA | Є
B➔ b | bS | aBB

Given string babaababbaa, shows leftmost and rightmost derivation.

Leftmost derivation:

$S \Rightarrow_{lm}$ ba

$S \Rightarrow_{lm}$ bS; rule $A \Rightarrow$ aS
$S \Rightarrow_{lm}$ babA; rule $S \Rightarrow$ ba
$S \Rightarrow_{lm}$ babbAAS; rule $A \Rightarrow$ bAA
$S \Rightarrow_{lm}$ babbaAS; rule $A \Rightarrow$ a
$S \Rightarrow_{lm}$ babbaaS; rule $A \Rightarrow$ aS
$S \Rightarrow_{lm}$ babbaabA; rule $A \Rightarrow$ bA
$S \Rightarrow_{lm}$ babbaabaS; rule $A \Rightarrow$ aS
$S \Rightarrow_{lm}$ babbaababA; rule $A \Rightarrow$ bA
$S \Rightarrow_{lm}$ babbaababbAA; rule $A \Rightarrow$ bAA
$S \Rightarrow_{lm}$ babbaababbaA; rule $A \Rightarrow$ a
$S \Rightarrow_{lm}$ babbaababbaa; rule $A \Rightarrow$ a

<u>Rightmost Derivation:</u>

$S \Rightarrow_{rm}$ bA
$S \Rightarrow_{rm}$ baS; rule $A \Rightarrow$ a S
$S \Rightarrow_{rm}$ babA; rule $S \Rightarrow$ bA
$S \Rightarrow_{rm}$ babbAA; rule $A \Rightarrow$ bAA
$S \Rightarrow_{rm}$ babbAa; rule $A \Rightarrow$ a
$S \Rightarrow_{rm}$ babbaSa; rule $A \Rightarrow$ a S
$S \Rightarrow_{rm}$ babbaaBa; rule $S \Rightarrow$ a B
$S \Rightarrow_{rm}$ babbaabSa; rule $B \Rightarrow$ b S
$S \Rightarrow_{rm}$ babbaabaBa; rule $S \Rightarrow$ a B
$S \Rightarrow_{rm}$ babbaababSa; rule $B \Rightarrow$ b S
$S \Rightarrow_{rm}$ babbaababbAas; rule $S \Rightarrow$ b A
$S \Rightarrow_{rm}$ babbaababbaa; rule $A \Rightarrow$ a

Direct Derivation:
$\alpha_1 \Rightarrow \alpha_2$: If α_2 can be derived directly from α_1, then it is direct derivation.
$\alpha_1 \Rightarrow^* \alpha_2$: If α_2 can be derived from α_1 with zero or more steps of the derivation, then it is derivation.

Example

$S \Rightarrow$ aSa | ab| a| b| Є

Direct derivation:

$S \Rightarrow$ ab
$S \Rightarrow$ aSa
$S \Rightarrow$ aaSaa
$S \Rightarrow$ aaabaa

Thus $S \Rightarrow^*$ aaabaa is just a derivation.

<u>**Language of Grammar (Context Free Grammar):**</u>

Let G = (V, T, P and S) is a context free grammar. Then the language of G denoted by L (G) is the set of terminal strings that have derivation from the start symbol in G.

i.e. $L(G) = \{x \,\varepsilon\, T^* \mid S \Rightarrow^* x\}$

The language generated by a CFG is called the Context Free Language (CFL).

<u>Derivation Tree / Parse Tree:</u>

Parse tree is a tree representation of strings of terminals using the productions defined by the grammar. A parse tree pictorially shows how the start symbol of a grammar derives a string in the language.

Parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding the replacement order.

Formally, given a Context Free Grammar G = (V, T, P and S), a parse tree is a n-ary tree having following properties:

- ► The root is labeled by the start symbol.
- ► Each interior node of parse tree are variables.
- ► Each leaf node of parse is labeled by a terminal symbol or Є.
- ► If an interior node is labeled with a non terminal A and its childrens æ x_1, x_2.................... x_n from left to right there is a production P as: A➔ x_1, x_2,......................... x_n for each x_i ε T.

Consider

S➔ aSa | a | b| Є

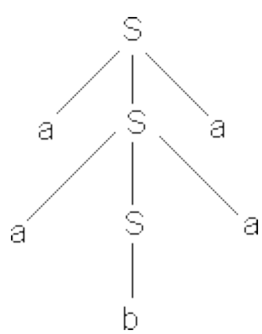
Now for string S➔* aabaa

We have, S➔ aSa

S➔aaSaa

S➔ aabaa

So the parse tree is:



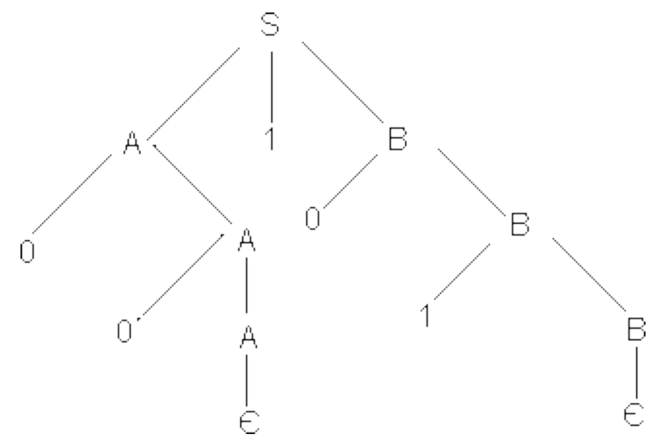
Consider the Grammar G:

S➔ A1B

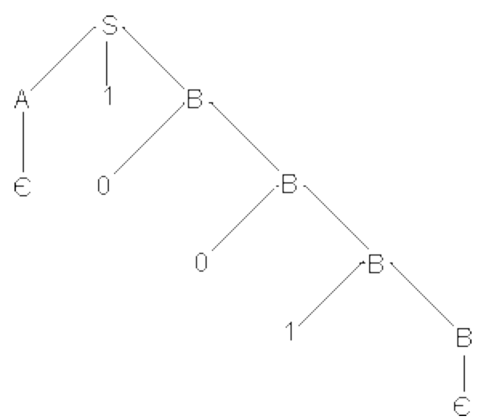
A➔ 0A | Є

B➔ 0B | 1B | Є

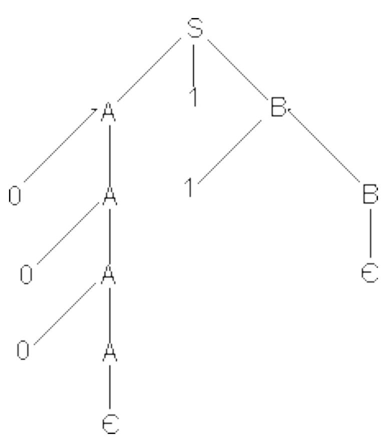
#Construct the parse tree for 00101:



#Construct parse tree for 1001:



#Construct parse tree for 00011:

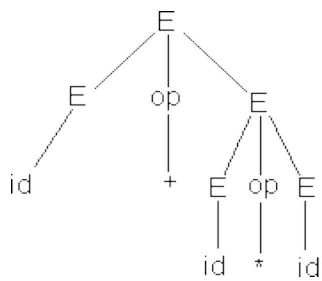


#Construct a Grammar defining arithmetic expression:

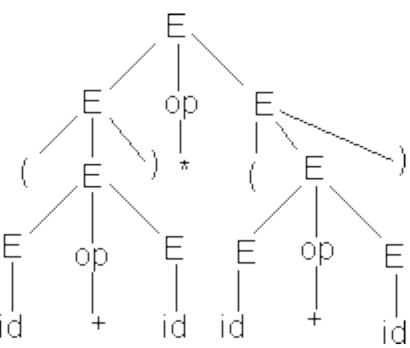
E➔ EOPE | (E) |id

OP➔ + | - | * | / | ↑

#Construct the parse tree for id + id *id



#Construct the parse tree for (id +id)*(id + id)



#Construct a CFG that generates language of balanced parentheses:

-Show the parse tree computation for

-( ) ( )

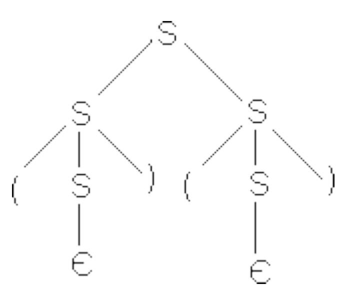
-((( ))) ( )

The CFG is

S➔ SS
S➔ (S)
S➔ Є
Where,
V = {S}
S = {S}
T = {(,)}
& P = as listed above

Now, for ( ) ( ),

S➔ SS
S➔ (S) S
S➔ (Є) S
S➔ (Є) (S)
S➔ (Є) (Є)
S➔ ( ) ( )



Similarly for ((( ))) ( ):

Ambiguity in Grammar:

A Grammar G = (V, T, P and S) is said to be ambiguous if there is a string w ε L (G) for which we can derive two or more distinct derivation tree rooted at S and yielding w.
In other words, a grammar is ambiguous if it can produce more than one leftmost or more than one rightmost derivation for the same string in the language of the grammar.
e.g.

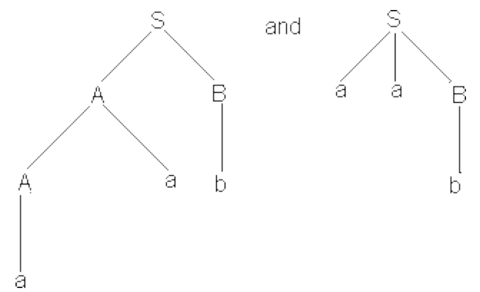
S➔ AB | aaB
A➔ a | Aa
B ➔ b

For any string aab;
We have two leftmost derivations as;

S➔ AB ➔ AaB
➔ aaB
➔ aab

Also, S➔ aaB ➔ aab

i.e.



Specification of CFG

The goal of this section is to show that every context free language (without Є) is generated by a CFG in which all production are of the form A➔ BC or A➔ a, where A, B and c are variables, and a is terminal. This form is called Chomsky Normal Form. To get there, we need to make a number of preliminary simplifications, which are themself useful in various ways;

- We must eliminate "useless symbols": Those variables or terminals that do not appear in any derivation of a terminal string from the start symbol.
- We must eliminate "Є-production": Those of the form A➔ Є for some variable A.
- We must eliminate "unit production": Those of the form A➔ B for variablesA and B.

1) Eliminating Useless Symbols: We say a symbol x is useless for a grammar G = (V, T, P, S) if there is some derivation of the form S ➔* α × β ➔*w, where w is in T*. Here, x may be either variable or terminal and the sentential form α × β might be the first or last in the derivation. If x is not useful, we say it is useless.

Thus useless symbols are those variables or terminals that appear in any derivation of a terminal string from the start symbol.

Eliminating a useless symbol includes identifying whether or not the symbol is "generating" and "reachable".

- ❖ We say x is generating if x➔*w for some terminal string w: Note that every terminal is generated since w can be that terminal itself, which is derived by zero steps.
- ❖ We say x is reachable if there is derivation S ➔* α × β for some α and β.

Thus if we eliminate the non generating symbols and then non-reachable, we shall have only the useful symbol left.

- ❖ For eliminating non generating symbol, identifying the non-generating symbols in CFG and eliminating those productions which contain non-generating symbols.

- For eliminating non-reachable symbols in CFG and eliminating those productions which contain non-reachable symbols.

Consider a grammar defined by following productions:

S➔ aB | bX
A➔ Bad | bSX | a
B➔ aSB | bBX
X➔ SBd | aBX | ad

Here;

A and X can directly generate terminal symbols. So, A and X are generating symbols. As we have;

A➔ a and X➔ ad

Also,

S➔ bX and X generates terminal string so S can also generate terminal string.

Hence, S is also generating symbol.

B can not produce any terminal symbol, so it is non-generating. Hence, the new grammar after removing non-generating symbols is as;

S➔ bX
A➔ bSX | a
X➔ ad

Here,

A is non-reachable as there is no nay derivation of the form S➔* α A β in the grammar.

Thus eliminating the non-reachable symbols, the resulting grammar is:

S➔ bX
X➔ ad

This is the grammar with only useful symbols.

Simple algorithm for removing non-generating symbols:

```
Begin
        Old value = Φ
        New value = {A| A ➔ W and W ε T*}
        While old value ≠ new value
                do
                        begin
                                Old value = new value
                                New value = old value U {A | A ➔ α for some α in
                                (T* U old value)
                        end
end
```

eg:

S→ A | B
A→aB | bS | b
B→ AB | Ba
C→ AS | b

Now using algorithm

Old value	New value
Φ	A, C
A, C	A, C, S
A, C, S	A, C, S

This will be final generating symbol.

Remove useless symbol from the following grammars:

1) S→ xyZ | XyzZ
X→Xz | xYZ
Y→ yYy | XZ
Z→ Zy | z

Here, Z can generate the terminal symbol so Z is generating symbol.

Similarly S→ xyZ →xyz. So S is also generating symbol as it can generate terminal symbol.

However, X and Y are non-generating as they can not generate terminal strings. So, removing the non-generating symbols, the non-generating symbols, the resulting grammar is:

S → xyZ
Z → Zy | z

Here, are no any non-reachable symbols. Hence is final grammar with all useful symbols.

2) S→ aC | SB
A→ bSCa
B→ aSB | bBC
C→ aBc | ad

Here, C→ ad,

S→ aC
→ aad and
A→ bSCa
→ bSada
→ baCada
→ baadad

Hence, C, S and A are generating symbols. While B can not generate any terminal string. So removing non-generating symbols;

S→aC
A→ bSCa
C→ ad

Here, A is non-reachable. So removing it

$S \rightarrow aC$
$C \rightarrow ad$ is final grammar with all useful symbols.

2) Eliminating Є-productions:

A grammar is said to have Є-productions if there is a production of the form $A \rightarrow Є$. Here our strategy is to begin by discovering which variables are "nullable". A variable 'A' is "nullable" if $A \rightarrow^* Є$.

<u>Algorithm:</u>

- ✓ If there is a production of the form $A \rightarrow Є$, then A is "nullable".
- ✓ If there is production of the form $B \rightarrow X_1, X_2$............ And each X_i's are nullable then B is also nullable.
- ✓ Find all the nullable variables.
- ✓ If $B \rightarrow X_1, X_2$.................... X_n is a production in P and then add all productions P' formed striking out some subsets of there X_i's that are nullable.
- ✓ Do not include $B \rightarrow Є$ if there is such production.

Consider the grammar:

$S \rightarrow ABC$
$A \rightarrow BB \mid Є$
$B \rightarrow CC \mid a$
$C \rightarrow AA \mid b$

Here,

$A \rightarrow Є$	A is nullable.
$C \rightarrow AA \rightarrow^* Є$	C is nullable
$B \rightarrow CC \rightarrow^* Є$	B is nullable
$S \rightarrow ABC \rightarrow^* Є$	S is nullable

Now for removal of Є –production;

$S \rightarrow ABC \mid AB \mid BC \mid AC \mid A \mid B \mid C$
$A \rightarrow BB \mid B$
$B \rightarrow CC \mid C \mid a$
$C \rightarrow AA \mid A \mid b$

Here, we add the production formed by striking out some subsets of the nullable symbols. Such as;

In production

$S \rightarrow ABC$, all A, B and C are nullable. So, striking out subset of each the possible combination of production is added as above.

Also, we eliminate all productions of the form $X \rightarrow Є$.

Remove Є-productions for each of grammar;

1) $S \rightarrow AB$
$A \rightarrow aAA \mid Є$
$B \rightarrow bBB \mid Є$

Here,

A➔ Є A is nullable
B➔ Є B is nullable.
S➔ AB ➔* S is nullable.

So, removing Є-productions.

S➔ AB | A | B
A➔ aAA | aA | a
B➔ bBB | bB | b

2) S➔ a | Xb | aYa
X➔ Y | Є
Y➔ b | Є

Here, X and Y are nullable. So, removing Є-productions.

S➔ a | Xb | aYa | b | aa
X➔ Y
Y➔ X | b

3) S➔ AACD
A➔ aAB | Є
C ➔ aC | a
D➔ aDa | bDb | Є

Here, A and D are nullable. So, removing Є-productions.

S➔ AACD | ACD | AAC | AC | C
A➔ aAb | ab
C➔ aC | a
D➔ aDa |abDb | bb

4) Eliminating Unit Production:

A unit production is a production of the form A➔ B, where A and B are both variables.

Here, if A➔ B, we say B is A-derivable.
B➔ C, we say C is B-derivable.
Thus if both of two A➔ B and B➔ C, then
A➔* C, hence C is also A-derivable.
Here pairs (A, B), (B, C) and (A, C) are called the unit pairs.

To eliminate the unit productions, first find all of the unit pairs.

The unit pairs are;

- (A, A) is a unit pair for any variable A as A➔* A
- If we have A➔ B then (A, B) is unit pair i.e. (A, B) is unit pair.
- If (A, B) is unit pair i.e. A➔B, and if we have B➔ C then (A, C) is alsoa unit pair.

Note:

- If (A, B) is a unit pair, then A➔ B means that B is A-derivable.
- If B is A-derivable and B➔ C then C is also A-derivable.

Now, to eliminate those unit productions for a, gives grammar say G = (V, T, P, S), we have to find another grammar G' = (V, T, P', S) with no unit productions. For this, we may workout as below;

- Initialize P' = P
- For each A ε V, find a set of A-derivable variables.
- For every pair (A, B) such that B is A-derivable and for every non-unit production B➔ α, we add production A ➔ α is P' if it is not in P' already.
- Delete all unit productions from P'.

Remove the unit production for grammar G defined by productions:

P = {S➔ S + T | T
T➔ T* F | F
F➔ (S) | a
};

➔P' = {S➔ S + T | T
T➔ t*F | F
F➔ (S) | a

Now, find unit pairs;

Here, S➔ T	So, (S, T) is unit pair.
T➔ F	So, (T, F) is unit pair.
Also, S➔T and T➔ F	So, (S, F) is unit pair.

Now, add each non-unit productions of the form
B ➔ α for each pair (A, B);

P' = {S➔ S + T |T * F| (S) | a
T➔T * F | (S) | a | F
F➔ (S) | a

Delete the unit productions from the grammar;

P' = {S➔ S + T | T * F | (S) | a
T➔ T * F | (S) | a
F➔ (S) | a

Simply the grammar G = (V, T, P, S) defined by following productions.

S➔ ASB | Є
A➔ aAS | a
B➔ SbS | A | bb | Є

Removing Є-productions;

Here, S➔ Є and B➔ Є So S and B are nullable.

Now, removing Є-productions, it yields;

S➔ ASA | AS | AB | A
A➔ aSA | aA | a
B➔ SbS | Sb | bS | b | A | bb

Removing unit productions;

Here, S➔ A, B➔ A. So, (S, A) and (B, A) are two unit pairs.
Hence, removing unit productions, we have;

S➔ ASB | AS | AB | aSA | aA | a
A➔ aSA | aA | a
B➔ SbS | Sb | bS | b | bb | aAS | aA | a

This grammar has no useless symbols. Hence it is the final simplified grammar.

S➔ 0A0 | 1B1 | 00 | 11| BB
A➔ 0A0 | 1B1 | 00 | 11 | BB
B➔ 0A0 | 1B1 | 00 | 11 | BB
C➔ 0A0 | 11 | 00 | 1B1 | BB

Removing useless symbols;

Here, C is useless as it is non-reachable symbol. So, removing it, simplified version of grammar is

S➔ 00 | 11 | 0A0 | 1B1 | BB
A➔ 00 | 11| 0A0 | 1B1| BB
B➔ 00 |11| 0A0 | 1B1 | BB

Simplify the grammar defined by following production:

S➔ 0A0 | 1B1 | BB
A➔ C
B➔ S | A
C➔ S | Є

➔ Now for removing the Є- productions:

Here, C➔ Є So, C is nullable
A➔ C and C➔ Є So, a is nullable.
B➔A, A➔ C, C➔ Є i.e. B➔* Є Hence B is nullable.

Also, S➔ BB and B is nullable hence S is nullable.

Now, removing Є-production, we have,

S➔ 0A0 | 00 | 1B1 | 11 | BB | B
A➔ C
B➔ S | A
C➔ S

Now, removing unit productions:

Here,

S➔ B, So, (S, B) is unit pair.
A➔ C So, (A, C) is unit pair.
B➔ S So, (B, S) is a unit pair.
B➔ A So, (B, A) is unit pair.
C➔ S So, (C, S) is unit pair.

Similarly, we find unit pairs as;
(C, A), (A, B), (S, C), (A, S), (S, A), (C, B), (B, C), (A, A), (B, B), (C, C), (S, S).

Chomsky Normal Form

A context free grammar G = (V, T, P, S) is said to be in Chomsky's Normal Form (CNF) if every production in G are in one of the two forms;

A➔ BC and

A➔ a where A, B, C ε V and a ε T

Thus a grammar in CNF is one which should not have;

- Є-production
- Limit production
- Useless symbols.

Theorem: Every context free language (CFL) without Є-production can be generated by grammar in CNF.

Proof:

If all the productions are of the form A➔ a and A➔ BC with A, B, C ε V and a ε T, we have them as it is. However if the productions are of the form:

A➔ X_1, X_2, ………………X_m, m>2 and if some X; is terminal a, then we replace the X_i by C_a having C_a➔ a where C_a is a variable itself.

Thus we have all productions of the form:

A➔ B_1B_2…………B_m, m>2; where all B_i's are non-terminal.

Then, we replace A➔ B_1B_2…………B_m, m>2by

A➔ B_1D_1; D_1➔ B_2B_3………………B_m, m>2

D_1➔ B_2D_2; D_2➔ B_3B_4………………B_m, m>2

And so on ………………….

Finally, all of the productions are achieved in the form as:

A➔ BC or A➔ a

This is certainly a grammar in CNF and generates a language without Є-productions.

Consider an example:

S➔ AAC
A➔ aAb | Є
C ➔ aC | a

1) Now, removing Є- productions;

Here, A is nullable symbol as A➔ Є

So, eliminating such Є-productions, we have;

S➔ AAC | AC | C
A➔ aAb | ab
C➔ aC | a

2) Removing unit-productions;

Here, the unit pair we have is (S, C) as S➔ C

So, removing unit-production, we have;

S➔ AAC | AC | C
A➔ aAb | ab

$C \rightarrow aC \mid a$

Here we do not have any useless symbol. Now, we can convert the grammar to CNF. For this;

- First replace the terminal by a variable and introduce new productions for those which are not as the productions in CNF.

i.e.

$S \rightarrow AAC \mid AC \mid C_1C \mid a$
$C1 \rightarrow a$
$A \rightarrow C_1AB_1 \mid C_1B_1$
$B_1 \rightarrow b$
$C \rightarrow C_1C \mid a$

Now, replace the sequence of non-terminals by a variable and introduce new productions.
Here, replace $S \rightarrow AAC$ by $S \rightarrow AC_2$, $C_2 \rightarrow AC$
Similarly, replace $A \rightarrow C_1AB_1$ by $A \rightarrow C_1C_3$, $C_3 \rightarrow AB_1$
Thus the final grammar in CNF form will be as;

$S \rightarrow AC_2 \mid AC \mid C_1C \mid a$
$A \rightarrow C_1C_3 \mid C_1b_1$
$C_1 \rightarrow a$
$B_1 \rightarrow b$
$C_2 \rightarrow AC$
$C_3 \rightarrow AB_1$
$C \rightarrow C_1C \mid a$

Simplify following grammars and convert to CNF;

1) $S \rightarrow ASB \mid \epsilon$
$A \rightarrow aAS \mid a$
$B \rightarrow SbS \mid A \mid bb$

2) $S \rightarrow AACD$
$A \rightarrow aAb \mid \epsilon$
$C \rightarrow aC \mid a$
$D \rightarrow aDa \mid bDa \mid \epsilon$

Solutions:

First remove ϵ-productions;
Here, $S \rightarrow \epsilon$ So, S is nullable.
So, the grammar becomes:

$S \rightarrow ASB \mid AB$
$A \rightarrow aAS \mid aA \mid a$
$B \rightarrow SbS \mid Sb \mid bS \mid A \mid bb$

Removing unit productions;
Here, $B \rightarrow A$ So, (B, A) is a unit pair.
Hence, removing this production yields;

$S \rightarrow ASB \mid AB$
$A \rightarrow aAS \mid aA \mid a$
$B \rightarrow Sb \mid bS \mid SbS \mid aAS \mid aA \mid a \mid b \mid bb$

Here, we have no useless symbols.

Now, to convert into CNF, replace each terminal by variables and introduce a new production as;

$S \rightarrow ASB \mid AB$
$A \rightarrow C_1AS \mid C_1A \mid a$
$C_1 \rightarrow a$
$B \rightarrow SB_1 \mid B_1S \mid SB_1S \mid C_1AS \mid C_1A \mid a \mid B_1B_1 \mid C_1b$
$B_1 \rightarrow b$

Also, replace $S \rightarrow ASB$ by $S \rightarrow AC_2$ with $C2 \rightarrow SB$
Replace $A \rightarrow C_1AS$ by $A \rightarrow C_1C3$ with $C_3 \rightarrow AS$
Replace $B \rightarrow SB_1S$ by $B \rightarrow SB_2$ with $B_2 \rightarrow B1S$
Also, $B \rightarrow C_1AS$ by $B \rightarrow C_1C_3$

So the grammar is;

$S \rightarrow AC_2 \mid AB$
$A \rightarrow C_1C_3 \mid C_1A \mid a$
$B \rightarrow SB_2 \mid SB_1 \mid B_1 \mid C_1C_3 \mid C_1A \mid B_1B_1 \mid b$
$C_1 \rightarrow a$, $C_2 \rightarrow SB$, $B_1 \rightarrow b$, $C_3 \rightarrow AS$, $B_2 \rightarrow B_1S$

Simplify the grammar and convert to CNF:

$S \rightarrow aaaaS \mid aaaa$

First we do not have any Є-production, useless symbol and unit productions.
So, converting it to the CNF, it consists;

- Replace a by any variable say A, with production

  $A \rightarrow a$

  So,

  $S \rightarrow AAAAS \mid AAAA$

Now, replace $S \rightarrow AAAAS$ by $S \rightarrow AA_1$ with $A_1 \rightarrow AAAS$
Again $A_1 \rightarrow AAAS$ by $A_1 \rightarrow AA_2$ with $A_2 \rightarrow AAS$
Again $A_2 \rightarrow AAS$ by $A_2 \rightarrow AA_3$ with $A_3 \rightarrow AS$
Similarly replace $S \rightarrow AAAA$ by $S \rightarrow AC$ with $C \rightarrow Aa$
and $C \rightarrow AAA$ by $C \rightarrow AC_1$ with $C_1 \rightarrow AA$

Thus, the grammar in CNF is;

$S \rightarrow AA1 \mid AC$
$A_1 \rightarrow AA_2$
$A_2 \rightarrow AA_3$
$A_3 \rightarrow AS$
$C \rightarrow AC_1$
$C_1 \rightarrow AA$
$A \rightarrow a$

Left recursive Grammar

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation ($A \Rightarrow^* A\alpha$) for some string x.
The top down parsing methods can not handle left recursive grammars, so a transformation that eliminates left recursion is needed.
Removal of immediate left recursion:

Let $A \Rightarrow A\alpha \mid \beta$, where β does not start with A.

Then, the left-recursive pair of productions could be replaced by the non-left-recursive productions as;

$A \Rightarrow \beta A'$
$A' \Rightarrow \alpha A' \mid \epsilon_x$; without changing the set of strings derivable from A.

No matter how many A-productions there are, we can eliminate immediate left recursion from them.
So, in general,

If $A \Rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots\ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$;
With βi does not start with A.

Then we can remove left recursion as:

$A \Rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots\ldots \mid \beta_n A'$
$A' \Rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots\ldots \mid \alpha_m A' \mid \epsilon$

Equivalently, these productions can be rewritten as:

$A \Rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots\ldots \mid \beta_n A' \mid \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$
$A' \Rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots\ldots \mid \alpha_m A' \mid \alpha_1 \mid \alpha_2 \mid \ldots\ldots \mid \alpha_m$

 # Consider an example

S→ AA | 0
A→ AAS | 0S | 1 *(Where AS is of the form = α_1, 0S of the form $=\beta_1$ and 1of the form$=\beta_2$)*

Here, the production A→ AAS is immediate left recursive. So removing left recursion, we have;

S→AA | 0
A→ 0SA' | 1A'
A' → ASA' | Є

Equivalently, we can write it as:

S→ AA | 0
A→ 0SA' | 1A' | 0S | 1A'
A'→ ASA' | AS

 # For the following grammar

E→ E+T |T
T→ T*F |F
F→ (E) | a

Here, E→ E+T and T→ T*F are left recursive so removing the left recursive

E→ TA'
A'→ +TA' | Є
T→ FB'
B'→ *FB' | Є
F→ (E) | a

Equivalently

$E \rightarrow TA' \mid T$
$A' \rightarrow +TA' \mid +T$
$T \rightarrow FB' \mid F$
$B' \rightarrow *FB' \mid *F$
$F \rightarrow (E) \mid a$

Greibach Normal Form (GNF)

A grammar G = (V, T, P and S) is said to be in Greibach Normal Form, if all the productions of the grammar are of the form:

$A \rightarrow a\,\alpha$, where a is a terminal, i.e. $a \in T^*$ and α is a string of zero or more variables. i.e. $\alpha \in V^*$

So we can rewrite as:

$A \rightarrow aV^*$ with $a \in T^*$
Or,
$A \rightarrow aV+$
$A \rightarrow a$ with $a \in T$

This form called Greibach Normal Form, after Sheila Greibach, Who first gave a way to construct such grammars. Converting a grammar to this form is complex, even if we simplify the task by, say, starting with a CNF grammar

To convert a grammar into GNF;

- Convert the grammar into CNF at first.
- Remove any left recursions.
- Let the left recursion tree ordering is $A_1, A_2, A_3, \ldots\ldots\ldots\ldots A_p$
- Let A_p is in GNF
- Substitute A_p in first symbol of A_{p-1}, if A_{p-1} contains A_p. Then A_{p-1} is also in GNF.
- Similarly substitute first symbol of A_{p-2} by A_{p-1} production and A_p production and so on…………

 # Consider an Example;

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

This Grammar is already in CNF

Now to remove left recursion, first replace symbol of A-production by S-production (since we do not have immediate left recursion) as:

$S \rightarrow AA \mid 0$
$A \rightarrow AAS \mid 0S \mid 1$ *(Where AS is of the form = α_1, 0S of the form $=\beta_1$ and 1of the form$=\beta_2$)*

Now, removing the immediate left recursion;

$S \rightarrow AA \mid 0$
$A \rightarrow 0SA' \mid 1A'$
$A' \rightarrow ASA' \mid \epsilon$

Equivalently, we can write it as:

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

A’ ➔ ASA’ | AS

Now we replace first symbol of S-production by A-production as;

S ➔ AASA’ | 1A’A | 0SA | 1A | 0

A ➔ 0SA’ | 1A’ | 0S | 1

A’ ➔ ASA’ | AS

Similarly replacing first symbol of A’-production by A-production, we get the grammar in GNF as;

S ➔ 0SA’A | 1A’A | 0SA | 1A | 0

A ➔ 0SA’ | 1A’ | 0S | 1

A’ ➔ 0SA’SA’ | 1A’SA’ | 0SSA’ | 1SA’| 0SA’S | 1A’S | 0SS | 1

Regular Grammar:

A regular grammar represents a language that is accepted by some finite automata called regular language. A regular grammar is a CFG which may be either left or right linear.
A grammar in which all of the productions are of the form A ➔ wB (*wB is oft the form α)* or A ➔ w for A, B ε V and w ε T* is called left linear.

Equivalence of Regular Grammar and Finite Automata

1. Let G = (V, T, P and S) be a right linear grammar of a language L(G). we can construct a finite automata M accepting L(G) as;

   M = (Q, T, δ, [S]. {[Є]})

   Where,

   Q – consists of symbols [α] such that α is either S or a suffix from right hand side of a production in P.

   T - is the set of input symbols which are terminal symbols of G.

   [S] – is a start symbol of G which is start state of finite automata.

   {[Є]} – is the set of final states in finite automata.

   And δ is defined as:

   If A is a variable then,

   δ([A], Є) = [α] such that A ➔ α is a production.

   If ‘a’ is a terminal and α is in T* U T*V then

   δ([Aα], a) = [α]

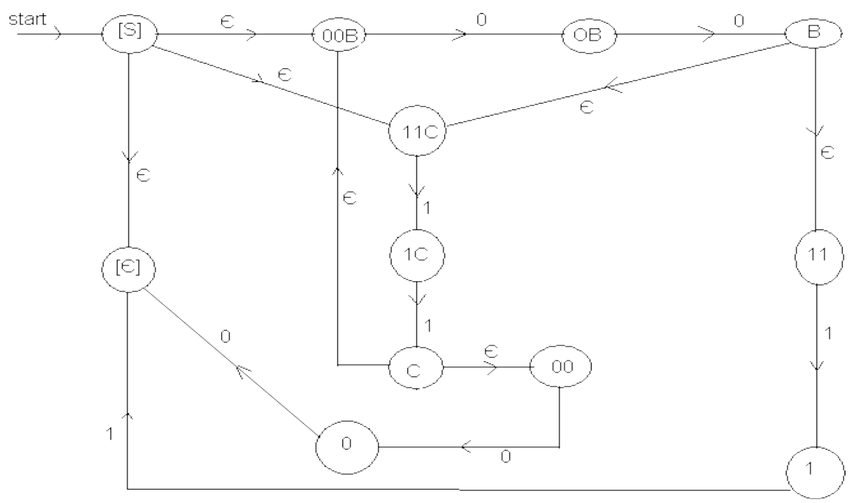
   For example;

   S ➔ 00B | 11C | Є

   B ➔ 11C | 11

   C ➔ 00B | 00

   Now the finite automata can be configured as;



2. If the regular grammar is of the form in which all the productions are in the form as;

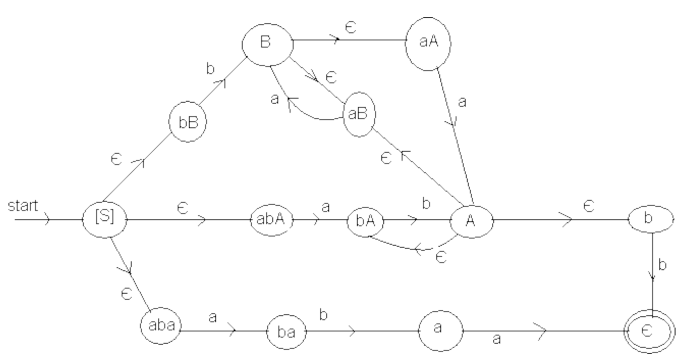
   A➔ aB or A➔ a, where A, B ε V and a ε T

   Then, following steps can be followed to obtain an equivalent finite automaton that accepts the language represented by above set of productions.

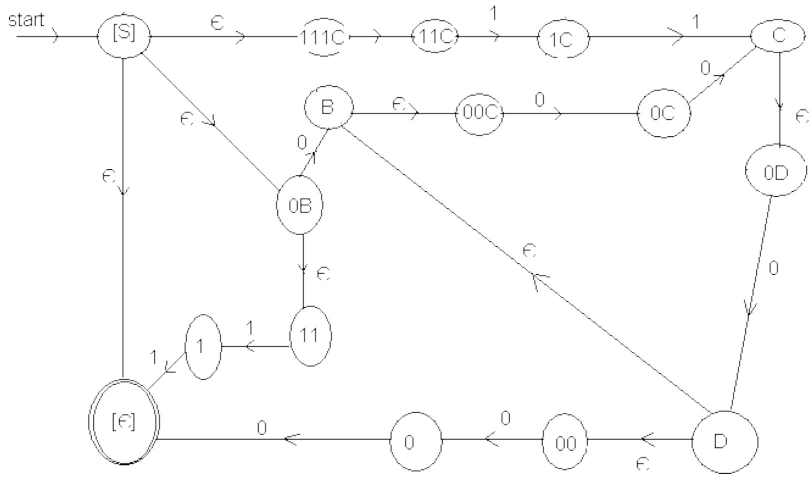
   a. The number of states in finite automaton will be one more than the number of variables in the grammar.(i.e. if grammar contain 4 variables then the automaton will have 5 states)
      - ► Such one more additional state in the final state of the automaton.
      - ► Each state in the automaton is a variable in the grammar.
   b. The start symbol of regular grammar is the start state of the finite automaton.
   c. If the grammar contains Є as S➔*Є, S being start symbol of grammar, then start state of the automaton will also be final state.

   Example;

   S➔ abA | bB | aba
   A➔ b | aB | bA
   B➔ aB | aA



S→ 0B | 111C | Є
B→ 00C | 11
C→ 00B | 0D
D→ 0B | 00

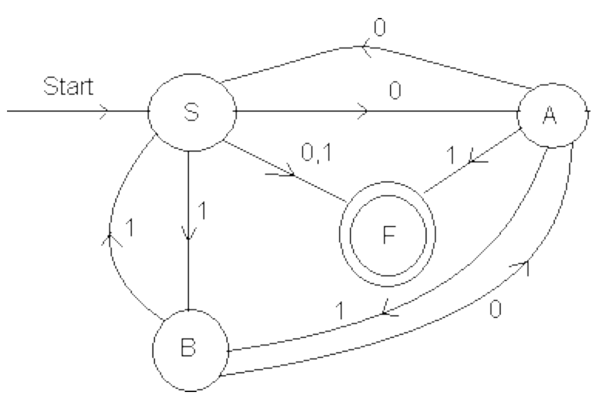


d. The transition function for the automaton is defined as;
► For each production A→ aB
δ(A, a) = B.
So there is an arc from state A to B labeled with a.
► For each production A→ a,
δ(A, a) = final state of automaton
► For each production A→ Є, make the state A as finite state.

Examples:

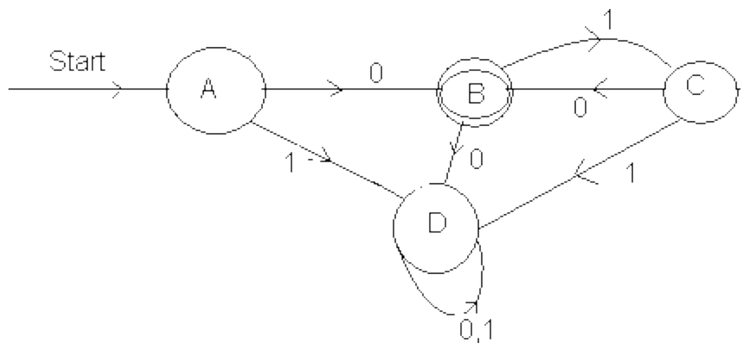
S→ 0A | 1B | 0 | 1
A→ 0S | 1B | 1
B→ 0A | 1S

The equivalent finite automaton can be configured as;



A→ 0B | 1D | 0
B→ 0D | 1C | Є
C→ 0B | 1D | 0
D→ 0D | 1D

The equivalent finite automaton can be configured as;



Construct equivalent finite automata for following regular grammar;

A→ 0B | E | 1C
B→ 0A | F | Є
C→ 0C | 1A | 0
E→ 0C | 1A | 1
F→ 0A | 1B | Є

Pumping lemma for context free Language:

The "pumping lemma for context free languages" says that in any sufficiently long string in a CFL, it is possible to find at most two short, nearby substrings that we can "pump" in tandem (one behind another). i.e. we may repeat both of the strings I times, for any integer I, and the resulting string will still be in the language. We can use this lemma as a tool for showing that certain languages are not context free.

Size of Parse tree: Our first step in pumping lemma for CFL's is to examine the size of parse trees, During the lemma, we will use the grammar in CNF form. One of the uses of CNF is to turn parses into binary trees. Any grammar in CNF produces a parse tree for any string that is binary tree. These trees have some convenient properties, one of which can be exploited by following theorem.

Theorem: Let a terminal string w be a yield of parse tree generated by a grammar G = (V, T, P and S) in CNF. If length path is n, then $|w| \leq 2^{n-1}$

Proof: Let us prove this theorem by simple induction on

Basis Step: Let n = 1, then the tree consists of only the root and a leaf labeled with a terminal.
So, string w is a simple terminal.

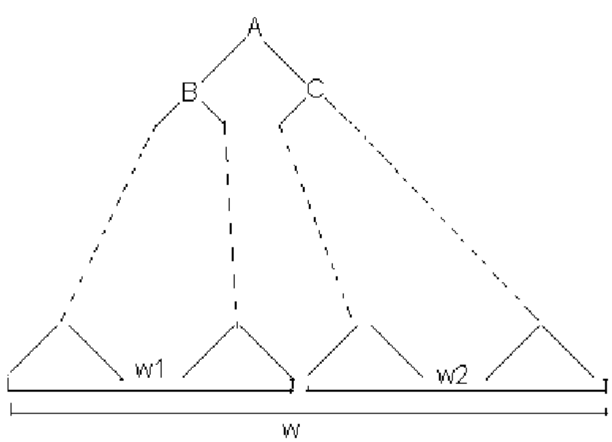
Thus $|w| = 1 = 2^{1-1} = 2^0$ which is true.

Inductive Step: Suppose n is the length of longest path and n>1. Then the root of the tree uses a production of the form; $A \rightarrow BC$, since n>1

No path is the sub-tree rooted at B and C can have length greater than n-1 since B&C are child of A. Thus, by inductive hypothesis, the yield of these sub-trees are of length almost 2^{n-1}.
The yield of the entire tree is the concatenation of these two yields and therefore has length at most,

$2^{n-2} + 2^{n-2} = 2^{n-1}$

Thus $|w| \leq 2^{n-1}$



Pumping Lemma

It states that every CFL has a special value called pumping length such that all longer strings in the language can be pumped. This time the meaning of pumped is a bit more complex. It means the string can be divided into five parts so that the second and the fourth parts may be repeated together any number of times and the resulting string still remains in the language.

Theorem: Let L be a CFL. Then there exists a constant n such that if z is any string in L such that |z| is at least n then we can write z=uvwxy, satisfying following conditions.

I) $|vx| > 0$

II) $|vwx| \leq n$

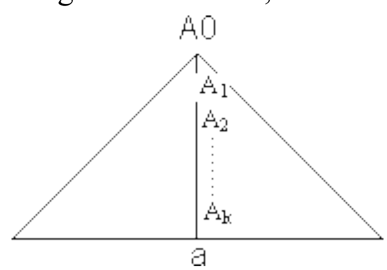
III) For all $i \geq 0$, $uv^iwx^iy \varepsilon L$. i.e. the two strings v & x can be "pumped" nay number of times, including '0' and the resulting string will be still a member of L.

Proof: First we can find a CNF grammar for the grammar G of the CFL L that generates L-{Є}.

Let m be the number of variables in this grammar choose $n= 2^m$. Next suppose z in L is of length at least n. i.e. $|z| \geq n$

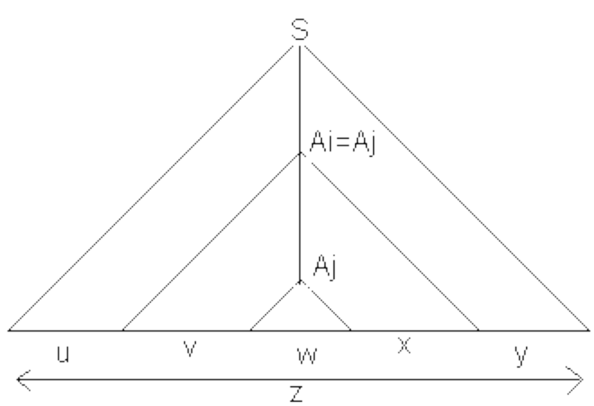
Any parse tree for z must have height m+1, other if it would be less than m+1, i.e. say m then by the lemma for size of parse tree, $|z| = 2^{m-1} = 2^m/2= n/2^m$ is contradicting. So it should be m+1.

Let us consider a path of maximum length in tree for z, as shown below, where k is the least m and path is of length k.

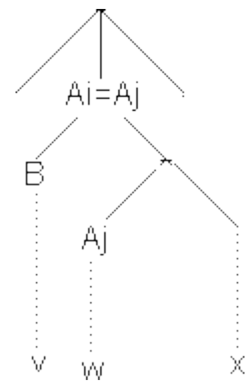


Since $k \geq m$. there are at least m+1 occurrence of variables A_0, A_1, A_2 .... A_k on the path. But there are only m variables in the grammar, so at least two of the last m+1 variable on the path must be same, (by pigeonhole principle).

Suppose $A_i = A_j$, then it is possible to divide tree



String w is the yield of sub-tree noted at A_j, string v and x are to left and right of w in yield of larger sub-tree rooted at Ai. If u and y represent beginning and end z i.e. to right and left of sub-tree rooted at A_i, then



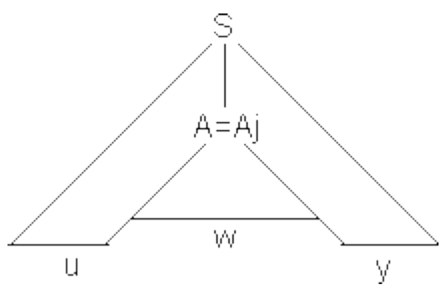
$$Z= uvwxy$$

Since, there are not unit productions so v & x both could not be Є, i.e. empty. Means that Ai has always two children corresponding to variables. If we let B denotes the one tree is not ancestor of A_j, then since w is derived from A_j then strings of terminal derived from B does not overlap x. it follows that either v or x is not empty. So,

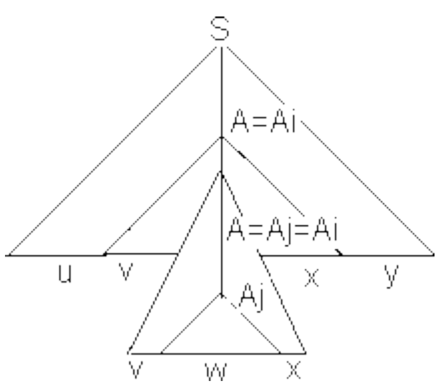
$$|vx| > 0$$

Now, we know Ai = Aj = A say, as we found any two variables in the tree are same.

Then, we can construct new parse tree where we can replace sub-tree rooted at Ai, which has yield vwx, by sub-tree rooted at Aj, which has yield w. the reason we can do so is that both of these trees have root labeled A. the resulting tree yields uv^0wx^0y as;



Another option is to replace sub-tree rooted at Aj by entire sub-tree rooted at Ai. Then the yield can be of pattern uv^2wx^2y



Hence, we can find any yield of the form $uv^iwx^iy \in L$ for any $i \geq 0$

Now, to show $|vwx| \leq n$. the Ai in the tree is the portion of path which is root of vwx. Since we begin with maximum path of length with height m+1. this sub-tree rooted at Ai also have height of less than m+1.

So by lemma for height of parse tree,

$$|vwx| \leq 2^{m+1-1} = 2^m = n$$

$$|vwx| \leq n$$

Thus, this completes the proof of pumping lemma.

Show that $L = \{a^n b^n c^n : n \geq 0\}$ is not a CFL

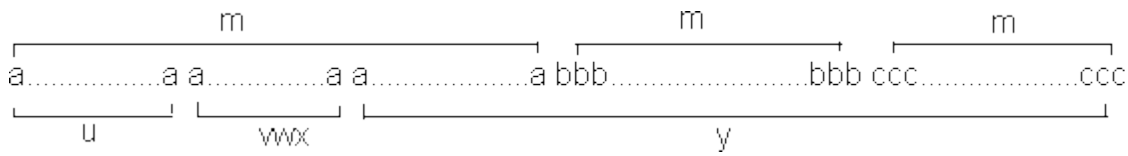
To prove L is not CFL, we can make use of pumping lemma for CFL. Let L be CFL.

Let any string in the language be; ambmcm

$z = uvwxy$, $|vwx| \leq m$, $|vx| > 0$

Case I:

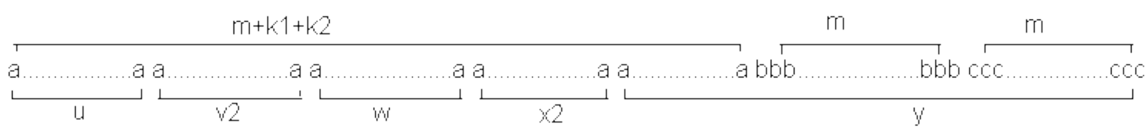
Vwx is in a^m



$v = a^{k1}$, $x = a^{k2}$; $k_1+k_2 > 0$ i.e. $k_1+k_2 \geq 1$

Now pump v and x

$m+ k_1+k_2$



But by pumping lemma, $uv^2xy^2z = a^{m+K1+k2}b^mc^m$ does not belong to L as, k_1+k_2.
Hence, it shows contradiction that L is CFL. So, L is not CFL.

Another way;

Case II: Suppose vwx overlaps $a^mb^mc^m$

i.e.

But here $|vwx| \leq m$ does not hold. So, the language is not CFL.

A context sensitive language is a formal language that can be defined by a context sensitive grammar.

Bakus-Naur Form

This is another notation used to specify the CFG. It is named so after John Bakus, who invented it, and Peter Naur, who refined it. The Bakus-Naur form is used to specify the syntactic rule of many computer languages, including Java. Here, concept is similar to CFG, only the difference is instead of using symbol "➔" in production, we use symbol ::=. We enclose all non-terminals in angle brackets, < >.

For example;

The BNF for identifiers is as;

<identifier> ::= <letter or underscore> | <identifier> | <symbol>

<letter or underscore> ::= <letter> | <_>
<symbol> ::= <letter or underscore> | <digit>
<letter> ::= a | b |.......................... | z
<digit> ::= 0 | 1| 2 |......................... |3

Context Sensitive Grammar

A context sensitive grammar (CSG) is a formal grammar in which left hand sides and right hand sides of any production rule may be surrounded by a context of terminal symbols. The concept of context sensitive grammar was introduced by N. Chomsky in 1950's as a way to describe the syntax of natural language.
Formally, CSG can be defined as;

A formal grammar G = (V, T, P and S) is context sensitive if all the productions, P, are of the form;

$\alpha A\beta \rightarrow \alpha\gamma\beta$, where $A \varepsilon V$, $\alpha, \beta \varepsilon (V \cup T)^*$ and $\gamma \varepsilon (V \cup T)^+$

The name context sensitive is explained by α and β that form the context of A and determine whether A can be replaced with γ or not.

e.g. $\{a^nb^nc^n \mid n \geq 1\}$ is a context sensitive language defined as;

S➔ aSBC | aBC
CB➔ BC
aB➔ ab
bB➔ bb
bc➔ bc
cC➔ cc

Closure Property of Context Free Languages

Given certain languages are context free, and a language L is formed from them by certain operation, like union of the two, then L is also context free. These theorem | lemma are often called closure properties of context free languages, since they show whether or not the class of context free language is called under the operation mentioned. Closure properties express the idea that one (or several) languages are context free, and then certain related languages are also context free or not.
Here are some of the principal closure properties for context free languages;

A. The context free language are closed under union

i.e. Given any two context free languages L_1 and L_2, their union $L_1 \cup L_2$ is also context free language.

Proof

The inductive idea of the proof is to build a new grammar from the original two, and from start symbol of the new grammar have productions to the start symbols of the original two grammars.

Let $G_1 = (V_1, T_1, P_1$ and $S_1)$ and $G_2 = (V_2, T_2, P_2$ and $S_2)$ be two context free grammars defining the languages $L(G_1)$ and $L(G_2)$. Without loss of generality, let us assume that they have common terminal set T, and disjoint set of non terminals i.e. V_1n_1. Because, the non-terminals are distinct so the productions P_1 and P_2.

Let S be a new non-terminal not in V_1 and V_2. Then, construct a new grammar G = (V, T, P, S) where;

$V = V_1 \cup V_2 \cup \{S\}$

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

G is clearly a context free grammar because the two new productions so added are also of the correct form, we claim that L (G) = L (G_1) U L (G_2). For this,

Suppose that x ε L (G_1). Then there is a derivation of x is

$S_1 \Rightarrow^* x$

But in G we have production,

$S \rightarrow S_1$

So there is a derivation of x also in G as:

$S \Rightarrow S_1 \Rightarrow^* x$

Thus, x ε L (G). Therefore, L (G_1) $\subseteq$ L (G)

A similar argument shows L (G_2) $\subseteq$ L (G)

So, we have,

$L(G_1) \cup L(G_2) \subseteq L(G)$

Conversely, suppose that x ε L (G). Then there is a derivation of x in G as:

$S \Rightarrow \beta \Rightarrow^* x$

Because of the way in which P is constructed, β must be either S_1 or S_2. Suppose β =S_1. Any derivation in G of the form $S_1 \Rightarrow^* x$ must involve only productions of G_1 so, $S_1 \Rightarrow^* x$ is a derivation of x in G_1.

Hence, β = S_1 ➔ x ε L(G_1). A simpler argument proves that β = S_1 implies x ε L (G_1). Thus L(G) $\subseteq$ L(G_1) U L.

It follows that L(G) = L(G_1) U L(G_2)

Similarly following two closure properties of CFL can be proved as for the union we have proved.

B. The CFLs are closed under concatenation

C. The CFLs are closed under kleen closure

However context free languages are not closed under some cases like, intersection and complementation.

D. The context free languages are not closed under intersection.

To prove this, the idea is to exhibit two CFLs whose intersection results in the language which is not CFL.

For this as we learned in pumping lemma that

$L = \{a^n b^n c^n \mid n \geq 1\}$ is not CFL

However following two languages are context free;

$L_1 = \{a^n b^n c^i \mid n \geq 1\ i \geq 1\}$

$L_2 = \{a^i b^n c^n \mid n \geq 1\ i \geq 1\}$

A grammar for L_1 is;

$S \rightarrow AC$

$A \rightarrow aAb \mid ab$

$C \rightarrow cC \mid c$

In this grammar, aA generates all strings of the form $a^n b^n$ and C generates all the strings of C's.

A grammar for L_2 is;

$S \rightarrow AB$

$A \rightarrow aA \mid a$

$B \rightarrow bBc \mid bC$

In this grammar, A generates any strings and B generates strings of the form $b^n c^n$.

Clearly,

$L = L_1 \cap L_2$. to see why, observe that L_1 requires that there be the same number of a's + b's, while L_2 requires the number of b's and c's to be equal. A string in both must have equal number of all three symbols and thus to be in L.

If the CFL's were closed under intersection, then we could prove the false statement that L is context free. Thus, we conclude by contradiction that the CFL's are not closed under intersection.

E. The CFL's are not closed under complementation.

To prove it, the idea is to show that if CFL's were closed under the operation of complementation, then they would also be closed under the operation of intersection, contradiction over property **D** .Let L_1 and L_2 be as defined in property then intersection of two languages can be expressed in term of complement as;

$L_1 \cap L_2 = \overline{(\overline{L1} \cup \overline{L2})} = \Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2))$

But we know CFL's are closed under union. So if CFL's were closed under complementation, then they would be also closed under intersection. Hence, it is not.