Unit IV
Context Free Grammar

Prepared By:
Ghanashyam BK

Introduction

- 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.
- CFG can define the languages that are regular as well as those languages that are not regular but cannot by RE.

Context Free Grammar

- A context free grammar is defined by 4-tuples (V, T, P,S) where,
  - V= set of variables
  - T= set of terminal symbols
  - P= set of rules and productions
  - S= start symbol and S ε V

Components of CFG

- **Terminal Symbols:**
  - a finite set of symbol that form the strings of language being defined.
  - it is represented by T
- **Variables**
  - a finite set of variables, also called sometimes non-terminals or non-terminal symbols or syntactic categories.
  - Each variable represents a language i.e. a set of strings.
- **Start symbol**
  - One of the variables represents the language being defined.

Components of CFG

- **set of productions or rules**
  - a finite set of productions or rules that represent the recursive definition of a language.
  - Each production consists of :
    - A variable that is being defined by the production. This is called head of production.
    - The production symbol →
    - A string of Zero or more terminals and Variables. This string is called the body of the production, represents one way to form the string in the language of the head.
- **Note:** 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.

Example

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
  - E → E Op E
  - E → int
  - E → (E)
  - Op → +
  - Op → -
  - Op → *
  - Op → /

Example

S → 0S1

S → ϵ

- This is a CFG defining the grammar of all the strings with equal no of 0's followed by equal no of 1's
- 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.

Compact Notation for the Productions

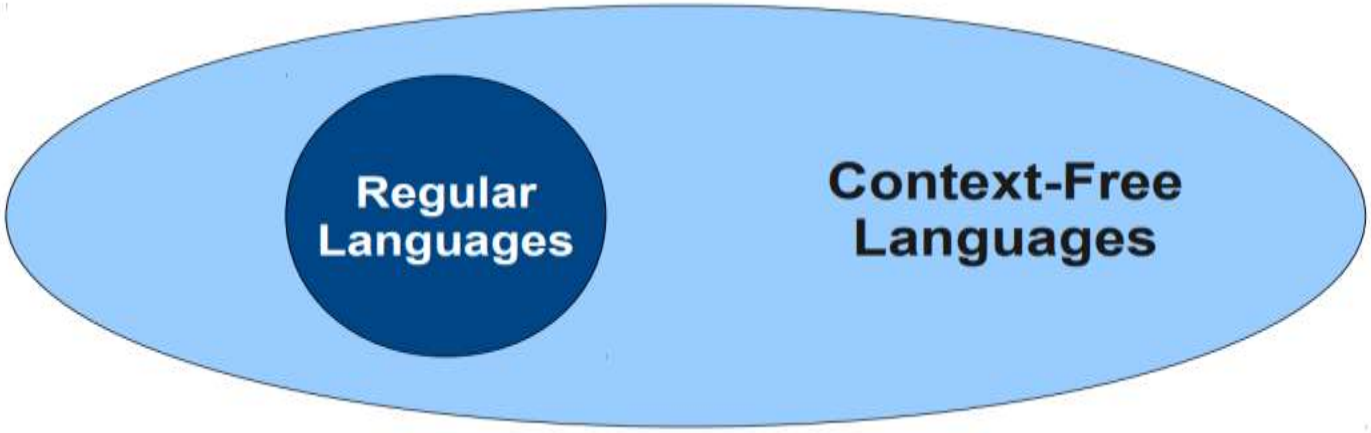
- CFG representing the language over $\Sigma = \{a, b\}$ which is palindrome language.
  - $S \rightarrow \epsilon \mid a \mid b$ .....Grammar (1)
  - $S \rightarrow a\ S\ a$
  - $S \rightarrow b\ S\ b$
- Here the first production is written in compact notation. The detail of this production look like as
  - $S \rightarrow \epsilon$
  - $S \rightarrow a$
  - $S \rightarrow b$

Use of CFG

- 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.
- Used in the development of the parser (determining if a string of tokens can be generated by a grammar.)

Context Free Language (CFL)

- A language generated by a CFG is called a context free language, abbreviated by CFL.
- CFL is a set of all the strings of terminals that can be produced from the start symbol S using the productions as substitutions.
- Every regular language is context-free, but not necessarily the other way around.



Exercise

- Construct a
  - CFG for palindromes with alphabets $\{0,1\}$.
  - CFG for language having any numbers of a's
  - CFG for any number of a's and b's
  - CFG for at least to a's
  - CFG for one occurrence of 000
  - CFG for $\{a^n b^n \mid n>=1\}$
  - CFG for $\{a^n b^{2n} \mid n>=1\}$

Derivation

- the process of deriving strings by applying productions rules of the CFG.
- 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 $0 \leq i \leq n$, the string $\beta_i\ \varepsilon\ (V \cup T)^*$
  - $B_0 = S$
  - For $0 \leq i \leq n$, there is a production of P that applied to β_i yields β_i+1
  - $B_n\ \varepsilon T^*$

Types of derivations

- **Body to head (Bottom Up) approach.**
  - 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 .....Grammer(1)
    - S → (S)
    - S → S-S
    - S → S*S
    - S → a
- Here given a + (a*a) / a – a

Types of derivations

- Now, by Body to head (Bottom Up) approach,

SN	String Inferred	Variable(For Language of)	Production	String Used
1	a	S	S→a	
2	a*a	S	S→S*S	String 1
3	(a*a)	S	S→(S)	String 2
3	(a*a)/a	S	S→S/S	String 3 and string 1
4	a+(a*a)/a	S	S→S+S	String 1 and 3
5	a+(a*a)/a-a	S	S→S-S	String 4 and 1

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

Types of derivations

- **Head to body (Top Down) approach.**
  - 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:** Here leftmost symbol (variable) is replaced first.
      Rightmost Derivation: Here rightmost symbol is replaced first.

Types of derivations

- For example: consider the previous example of deriving string **a+ (a*a)/a-a** with the grammar(1)
- Now leftmost derivation for the given string is:
  - S → S + S rule S → S + S
  - S → a + S rule S → a [Note here we have replaced left symbol of the body]
  - 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 " "

Types of derivations

- And rightmost derivation for **a+ (a*a)/a-a** 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; “

Direct Derivation:

- $\alpha 1 \rightarrow \alpha 2$: If $\alpha 2$ can be derived directly from $\alpha 1$, then it is direct derivation.
- $\alpha 1 \rightarrow^* \alpha 2$: If $\alpha 2$ can be derived from $\alpha 1$ with zero or more steps of the derivation, then it is just derivation.
- **Example**
  - S → aSa | ab| a| b| ϵ
  - **Direct derivation:**
    - S → ab
    - S → aSa
    - S → aaSaa
    - S → aaabaa
  - Thus S →* aaabaa is just a derivation.

Exercise

- The following grammar generates the language of regular expression 0*1(0+1)*:

  S → A1B

  A → 0A | ϵ

  B → 0B | 1B | ϵ

- Give the leftmost and rightmost derivation of the following strings:
  - 00101
  - 1001
  - 00011

Language of a grammar

- 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. $L(G) = \{x \in T^* \mid S \rightarrow^* x\}$

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

Parse Tree/Derivation Tree

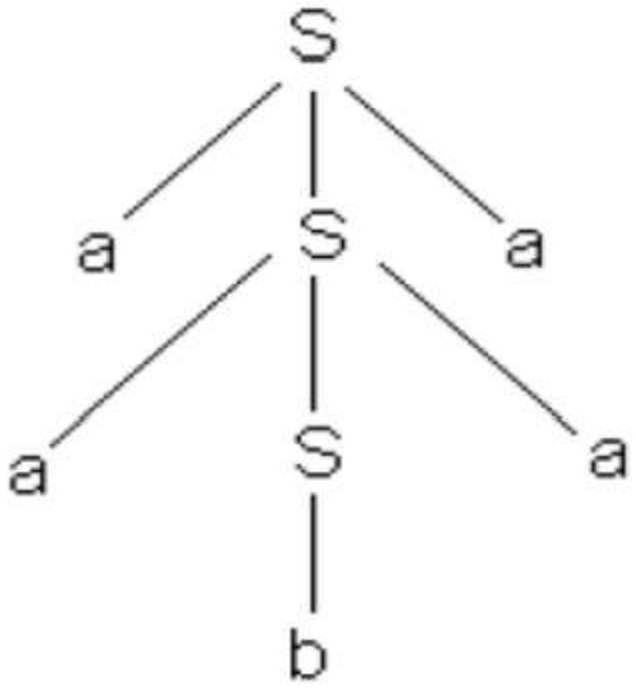
- a tree representation of strings of terminals using the productions defined by the grammar.
- A parse tree pictorially shows how the start symbol of a grammar derives a string in the language.
- may be viewed as a graphical representation for a derivation that filters out the choice regarding the replacement order.

Parse Tree/Derivation Tree

- 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 ϵ.
  - If an interior node is labeled with a non terminal A and its children's are $x_1, x_2 \ldots\ldots\ldots\ldots x_n$ from left to right then there is a production P as: $A \rightarrow x_1, x_2, \ldots\ldots\ldots\ldots\ldots x_n$ for each $x_i \in T$.

Parse Tree/Derivation Tree

- **Example**
- Consider the grammer
  - S → aSa | a | b| Є
- Now for string S→* aabaa
- We have,
  - S→ aSa
  - S→aaSaa
  - S→aabaa
  - So the parse tree is →:



Exercise

- Consider the Grammar G:

  S → A1B

  A → 0A | ϵ

  B → 0B | 1B | ϵ

  - Construct the parse tree for 00101
  - Construct the parse tree for 1001
  - Construct the parse tree for 00011

- Consider the grammer for the arithematic expression

  E → E OP E | (E) |id

  OP → + | - | * | /

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

Exercise

- Construct a CFG that generates language of balanced parentheses: -Show the parse tree computation for
  - ( ) ( )
  - ((( ))) ( )

- **Solution**
- The CFG is G=(V,T,P,S)
  - S → SS
  - S →(S)
  - S → ϵ
  - Where, V = {S}, S = {S}, T = {(,)} & P = as listed above
- Now find parse tree.

Ambiguous 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.

Ambiguous grammar

- **Example:**
  - S → AB | aaB
  - A → a | Aa
  - B → b
- For any string aab;
- We have two leftmost derivations as;
  - S → AB the parse tree for this derivation is:
  - S → AaB
  - S → aaB
  - S → aab
- Also,
  - S → aaB the parse tree for this derivation is:
  - S → aab

Exercise

- Find whether the following grammars are ambiguous or not.
  a) S → aS |Sa| Є
  b) E → E +E | E*E| id
  c) A → AA | (A) | a
  d) Grammar below
     - E → I
     - E → E + E
     - E → E * E
     - E → (E)
     - I → ε | 0 | 1 | ... | 9

Regular Grammars

- 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 $A \rightarrow wB$ (wB is of the form α) or $A \rightarrow w$ for A, B ε V and w ε T* is called **Right linear**.
- A grammar in which all of the productions are of the form $A \rightarrow Bw$ (Bw is of the form α) or $A \rightarrow w$ for A, B ε V and w ε T* is called **left linear**.

Regular Grammars

- **Examples:**
- The grammar G1= ({S}, {a, b}, S, P1), with P1 given as S →abS|a, It is right-linear.
- The Grammar G2 =({S,S1,S2}, {a, b}, S, P2) with productions S →S1ab, S1→S1ab|S2, S2 → a,It is left-linear.
- Both G1 and G2 are regular grammars.
- The grammar G= ({S, A, B},{a, b}, S, P), with production S→A, A→aB|λ, B→Ab. Is it a regular grammar?
  - It is not a regular grammar because it is neither right-liner not left-linear.

Equivalence of Regular Grammar and Finite Automata

- 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;

$$M = (Q, T, \delta, [S]. \{[\epsilon]\})$$

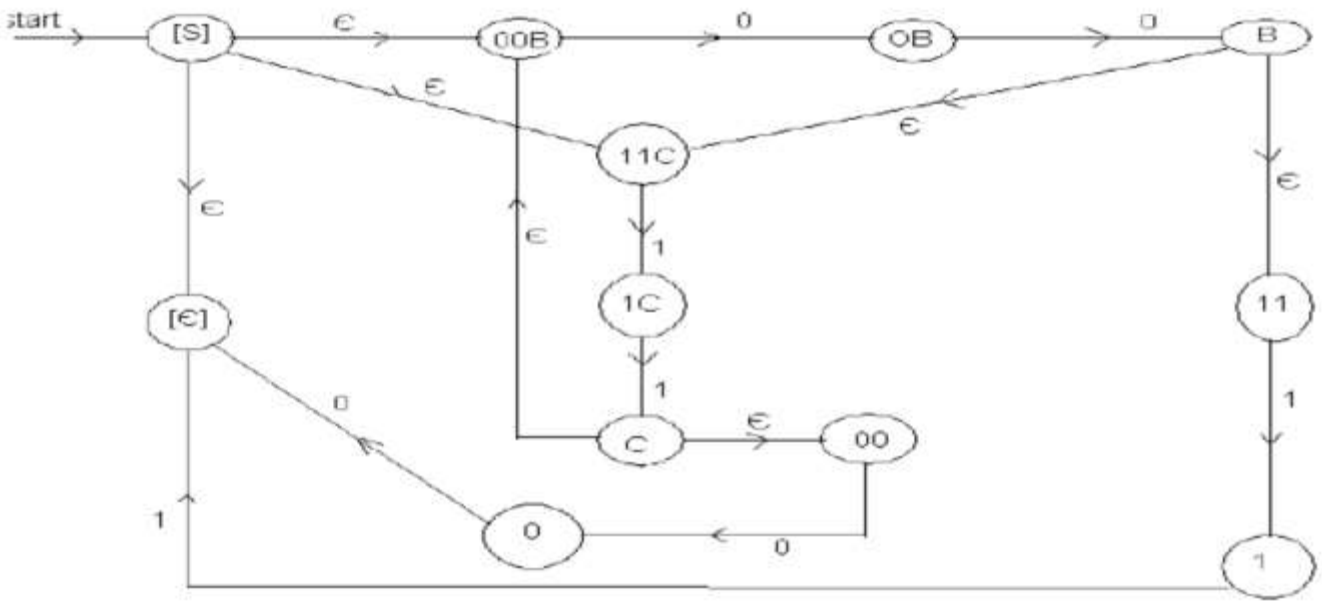
Where,

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

- And δ is defined as:
  - If A is a variable then, δ([A], ϵ) = [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].

Equivalence of Regular Grammar and Finite Automata

- For example;
  - S → 00B | 11C | Є
  - B → 11C | 11
  - C → 00B | 00
- Now the finite automata can be configured as;



Equivalence of Regular Grammar and Finite Automata

Another way:

- 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.
- 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.

Equivalence of Regular Grammar and Finite Automata

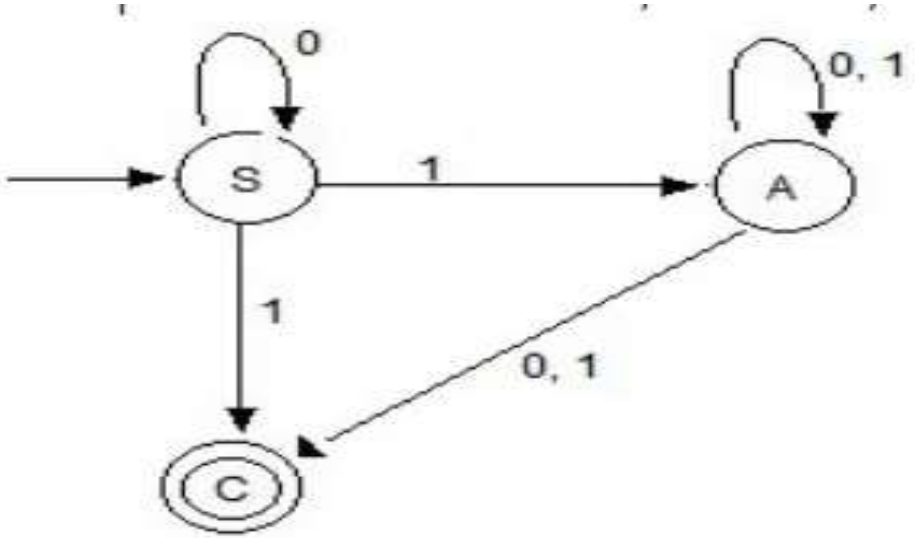
- The start symbol of regular grammar is the start state of the finite automaton.
- If the grammar contains ϵ as $S \rightarrow^* \epsilon$, S being start symbol of grammar, then start state of the automaton will also be final state.
- The transition function for the automaton is defined as;
  - For each production $A \rightarrow aB$, we write $\delta(A, a) = B$. So there is an arc from state A to B labeled with a.
  - For each production $A \rightarrow a$, $\delta(A, a)$ = final state of automaton
  - For each production $A \rightarrow \epsilon$, make the state A as finite state.

Examples

- Give the automaton for the following grammar:

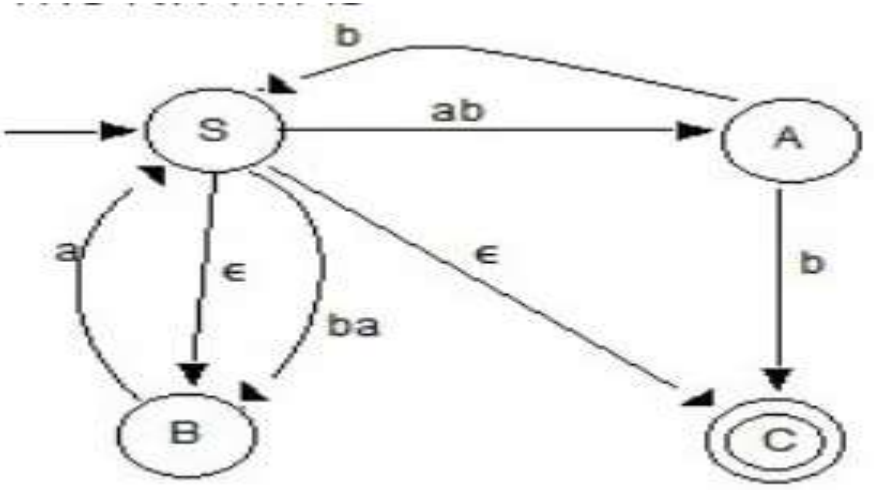
  S -> 0S/1A/1

  A -> 0A/1A/0/1



Examples

- Consider the regular grammar G = (V_n, V_t, P, S), where V_n = {A, B, S}, V_t = {a, b} and P is defined as
  S → abA
  S → B
  S → baB
  S → ϵ
  A → bS
  B → aS
  A → b
  Construct a NFA M such that L(M) = L(G).



Exercises

a) Write the equivalent finite automata

- S → abA | bB | aba
- A → b | aB | bA
- B → aB | aA

b) Write the equivalent finite automata

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

Exercises

a) Write the equivalent finite automata

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

b) Write the equivalent finite automata

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

Simplification of CFG

- 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 variables A and B.

Eliminating Useless Symbols

- useful 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**".
- **Generating Symbol:** We say x is generating 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.
- **Reachable symbol:** We say x is reachable if there is derivation S →* αxβ for some α and β.
- Thus if we eliminate the non generating symbols and then non-reachable, we shall have only the useful symbols left.

Eliminating Useless Symbols

Example:

- 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 the productions 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.

Eliminating Useless Symbols

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

  S→ bX

  A→ bSX | a

  X→ ad

- Here, A is non-reachable as there is no any derivation of the form $S \rightarrow^* \alpha A \beta$ 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.

Exercise

- Remove useless symbol from the following grammar:
  - S→ xyZ | XyzZ
  - X→Xz | xYZ
  - Y→yYy | XZ
  - Z→ Zy | z
- Remove useless symbol from the following grammar
  - S→ aC | SB
  - A→ bSCa
  - B→aSB | bBC
  - C→ aBc | ad

Eliminating Є-productions

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

Algorithm

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

Eliminating Є-productions

Example

- Consider the grammar:

  S→ABC

  A→ BB | Є

  B→ CC | a

  C→ AA | b

- Here,

A→ Є	A is nullable.
C→AA→* Є,	C is nullable
B→ CC→* Є,	B is nullable
S→ABC→* Є,	S is nullable

Eliminating Є-productions

- Now for removal of Є –production:
  - In production S→ABC, all A, B and C are nullable. So, striking out subset of each the possible combination of production gives new productions as:
    - S→ ABC | AB | BC | AC | A | B | C
- Similarly for other can be done and the resulting grammar after removal of Є-production is:
  - S→ABC | AB | BC | AC | A | B | C
  - A→BB | B
  - B→CC | C | a
  - C→AA | A | b

Exercise

- Remove Є-productions for each of grammar
  - S→ AB
  - A→ aAA | Є
  - B → bBB | Є

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.

Eliminating Unit Production

- 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 $A \rightarrow^* A$
  - If we have $A \rightarrow B$ then (A, B) is unit pair.
  - If (A, B) is unit pair i.e. $A \rightarrow B$, and if we have $B \rightarrow C$ then (A, C) is also a unit pair.

Eliminating Unit Production

- 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.
  - 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'.

Example

- Remove the unit production for grammar G defined by productions:
- P = { S→ S + T | T
  T→ T* F | F
  F→ (S) | a
  };

Initialize

- P' = { S→ S + T | T
  T→ T* F | F
  F→ (S) | a
  };

Example

- 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 | F | (S) | a
    F→ (S) | a
    };

Example

- Delete the unit productions from the grammar;
  - P' = { S → S + T | T * F | (S) | a
    T → T* F | (S) | a
    F → (S) | a
    };

Excercise

- Simply the grammar G = (V, T, P, S) defined by following productions.
  - S→ ASB | Є
  - A→ aAS | a
  - B→ SbS | A | bb | Є
- Simplify the grammar defined by following production:
  - S→ 0A0 | 1B1 | BB
  - A→ C
  - B→ S | A
  - C→S | Є

Chomsky Normal Form (CNF)

- 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
  - Unit 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 done.
- Otherwise, we have to do two task as:
  - Arrange that all bodies of length 2 or more consist only of variable
  - Break bodies of length 3 or more into a cascade of production , each with a body consisting of two variable.

Theorem

- The construction for task (1) is as follows
  - if the productions are of the form:
    - $A \rightarrow X_1, X_2, \ldots\ldots\ldots\ldots\ldots X_m$, $m>2$ and if some X_i is terminal a, then we replace the X_i by C_a having $C_a \rightarrow$ a where C_a is a variable itself.
  - Thus as result we will have all productions of the form:
    - $A \rightarrow B_1B_2\ldots\ldots\ldots\ldots B_m$, $m>2$; where all B_i's are non-terminal.

Theorem

- The construction for task (2) is as follows
  - We break those production $A \rightarrow B_1B_2.....B_m$ for m>=3, into group of production with two variables in each body.
  - We introduce m-2 new variables $C_1, C_2, ... C_{m-2}$. The original production is replaced by the m-1 productions
    - $A \rightarrow B_1C_1$,
    - $C_1 \rightarrow B_2C_2$,
    - .....
    - ....
    - $C_{k-2} \rightarrow B_{k-1}B_k$
  - Finally, all of the productions are achieved in the form as: $A \rightarrow BC$ or $A \rightarrow a$
- This is certainly a grammar in CNF and generates a language without ϵ-productions.

Example

S→ AAC

A→ aAb | ϵ

C → aC | a

- Now, removing ϵ- productions
- Here, A is nullable symbol as A→ϵ So, eliminating such ϵ-productions, we have;
  - S→AAC | AC | C
  - A→ aAb | ab
  - C→aC | a

Example

- Removing unit-productions
  - Here, the unit pair we have is (S, C) as S→ C
  - So, removing unit-production, we have;
    - S→ AAC | AC | aC| a
    - A→ aAb | ab
    - C→ aC | a
- Here we do not have any useless symbol. Now, we can convert the grammar to CNF. For this;

Example

- First replace the terminal by a variable and introduce new productions for those which are not as the productions in CNF. i.e.
  - $S \rightarrow AAC \mid AC \mid C_1C \mid a$
  - $C_1 \rightarrow a$
  - $A \rightarrow C_1AB_1 \mid C_1B_1$
  - $B_1 \rightarrow b$
  - $C \rightarrow C_1C \mid a$
- 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 $A \rightarrow C_1C_3$, $C_3 \rightarrow AB_1$

Example

- Thus the final grammar in CNF form will be as;
  - $S \rightarrow AC_2 \mid AC \mid C_1C \mid a$
  - $A \rightarrow C_1C_3 \mid C_1b_1$
  - $C_1 \rightarrow a$
  - $B_1 \rightarrow b$
  - $C_2 \rightarrow AC$
  - $C_3 \rightarrow AB_1$
  - $C \rightarrow C_1C \mid a$

Excercise

- Simplify following grammars and convert to CNF;
  - S→ASB | Є
  - A→ aAS | a
  - B→ SbS | A | bb
- Simplify following grammars and convert to CNF;
  - S→ AACD
  - A→ aAb | Є
  - C→ aC | a
  - D→aDa | bDa | Є
- Simplify following grammars and convert to CNF;
  - S→ aaaaS | aaaa

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 α) for some string α (which may variable or terminal).
- 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→ Aα | β, 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→ βA'
  - A'→ α A' | Є;
  - without changing the set of strings derivable from A.
- Or equvalnetly, these production can be rewritten as with out Є- production
  - A→ βA' | β
  - A'→ α A' | α;

Removal of immediate left recursion

- No matter how many A-productions there are, we can eliminate immediate left recursion from them
- So, in general,
  - If A→ A α1 | A α2 | ……| A αm | β1 | β2 | ……| βn; With βi does not start with A.
  - Then we can remove left recursion as:
    - A→ β1A' | β2A'| …… | βnA'
    - A'→ α1A' | α2A' | …… | αmA' | Є
  - Equivalently, these productions can be rewritten as:
    - A→β1A' | β2A'| …… | βnA' | β1 | β2 |……| βn
    - A'→α1A' | α2A' | …… | αmA' | α1 | α2| ……| αm

Example

- S→AA | 0
- A→ AAS | 0S | 1 (Where AS is of the form = α1, 0S of the form =β1 and 1of the form=β2)

- Here, the production A→ AAS is immediate left recursive. So removing left recursion,
- we have;
  - S→AA | 0
  - A→0SA' | 1A'
  - A' → ASA' | ϵ
- Equivalently, we can write it as:
  - S→ AA | 0
  - A→ 0SA' | 1A' | 0S | 1
  - A'→ ASA' | AS

Exercise

- For the following grammar

  E→E+T |T

  T→ T*F |F

  F→ (E) | a

  Remove the Left recursion.

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→ a α, where a is a terminal, i.e. a ε T* and α is a string of zero or more variables. i.e. α ε V*
  - So we can rewrite as:

    A→aV* with a ε T*

    OR

    A→ aV+

    A→ a with a ε T

Conversion of grammar into GNF

- Convert the grammar into CNF at first.
- Remove any left recursions.
- Let the left recursion tree ordering is $A_1, A_2, A_3, \ldots\ldots\ldots A_p$
- 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……

Example

S→AA | 0

A→SS | 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→ AA | 0

  A→ AAS | 0S | 1 (Where AS is of the form = α1, 0S of the form =β1 and 1of the form=β2)

Example

- Now, removing the immediate left recursion;

  S→AA | 0

  A→0SA' | 1A'

  A'→ ASA' | ϵ

- Equivalently, we can write it as:

  S→ AA | 0

  A→ 0SA' | 1A' | 0S | 1

  A'→ASA' | AS

Example

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

  S→0SA'A | 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 | 1S

Backus-Naur Form (BNF)

- 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 | …… | 9

Context Sensitive Grammar

- A Context-sensitive grammar is an Unrestricted grammar in which all the productions are of form

  $\alpha \rightarrow \beta$

  Where, $\alpha, \beta \in (V \cup T)^+$ and $|\alpha| <= |\beta|$

  - Where α and β are strings of non-terminals and terminals.
- Context-sensitive grammars are **more powerful** than context-free grammars because there are some languages that can be described by CSG but not by context-free grammars

Context Sensitive Language

- The language that can be defined by context-sensitive grammar is called CSL.
- **Example –**
  - Consider the following CSG.

    S ⟶ abc/aAbc

    Ab ⟶ bA

    Ac ⟶ Bbcc

    bB ⟶ Bb

    aB ⟶ aa/aaA
- The language generated by this grammar is $\{\mathbf{a^n b^n c^n} \mid n \geq 1\}$.

Pumping Lemma for CFL

- 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.

Pumping Lemma for CFL

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 n.
- **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.

Theorem

- **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→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-2} .
- The yield of the entire tree is the concatenation of these two yields and therefore has length at most,
  - $2^{n-2} + 2^{n-2} = 2^{n-1}$
  - $|w| <= 2^{n-1}$. Hence proved.

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.
- 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.
  - | vx| >0
  - |vwx| ≤ n
  - For all i ≥ 0, uv^iwx^iy ε L. i.e. the two strings v & x can be "pumped" any number of times, including '0' and the resulting string will be still a member of L.

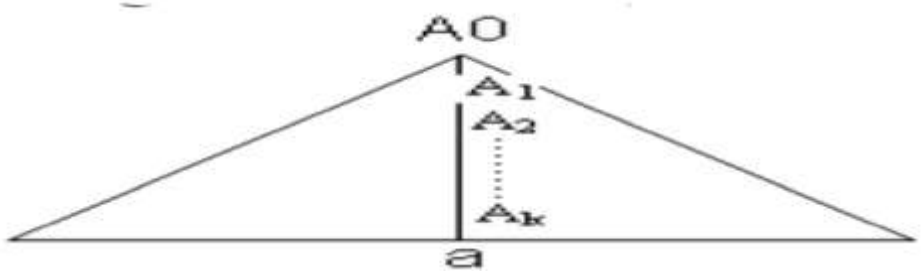
Theorem

Proof:

- First we can find a CNF grammar for the grammar G of the CFL L that generates L-
- 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, $|z| = 2^{m-1} = 2^m/2 = n/2$ is contradicting. So it should be m+1.

Theorem

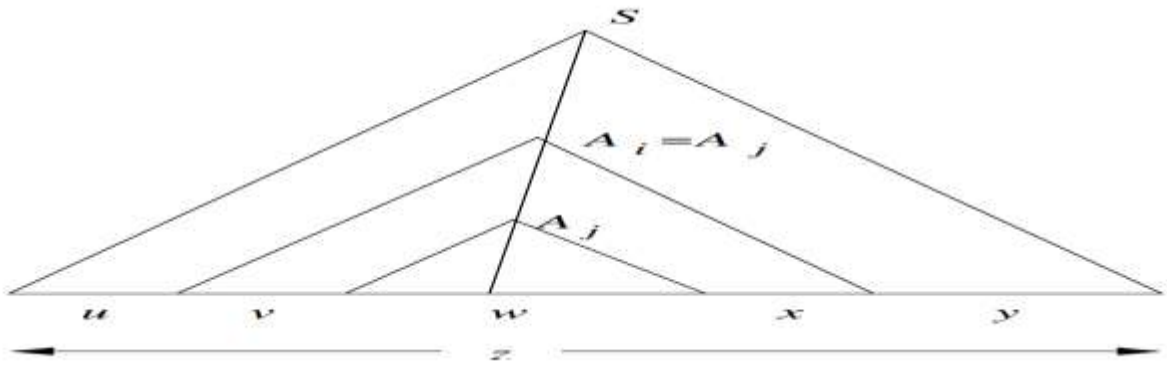
- Let us consider a path of maximum length in tree for z, as shown below, where k is the least m and path is of length k.



- Since k ≥ m. there are at least m+1 occurrence of variables A0, A1, A2..... Ak 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).

Theorem

- Suppose Ai = Aj, then it is possible to divide tree,



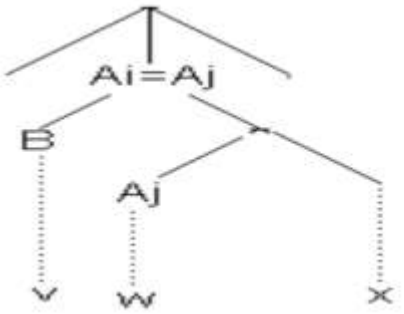
- String w is the yield of subtree noted at Aj, string v and x are to left and right of w in yield of larger subtree rooted at Ai.
- If u and y represent beginning and end z i.e. to right and left of sub-tree rooted at Ai, then
  - Z= uvwxy

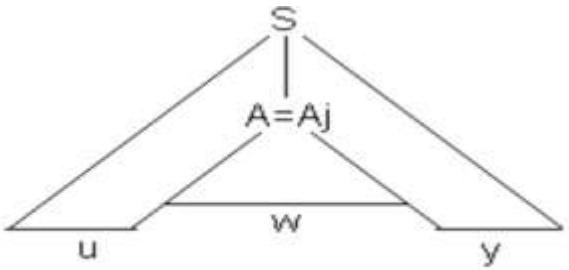
Theorem

- 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 Aj, then since w is derived from Aj then strings of terminal derived from B does not overlap x. it follows that either v or x is not empty.
  - So, $|vx| > 0$

Theorem

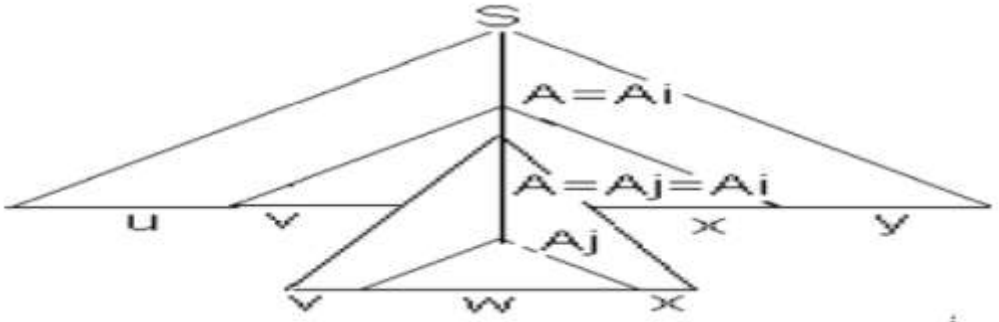
- Now, we know Ai = Aj = A say, as we found any two variables in the tree are same.
- Then, we can construct new parse tree where we can replace sub-tree rooted at Ai, which has yield vwx, by sub-tree rooted at Aj, which has yield w.
- the reason we can do so is that both of these trees have root labeled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





Theorem

- 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 as



- 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.

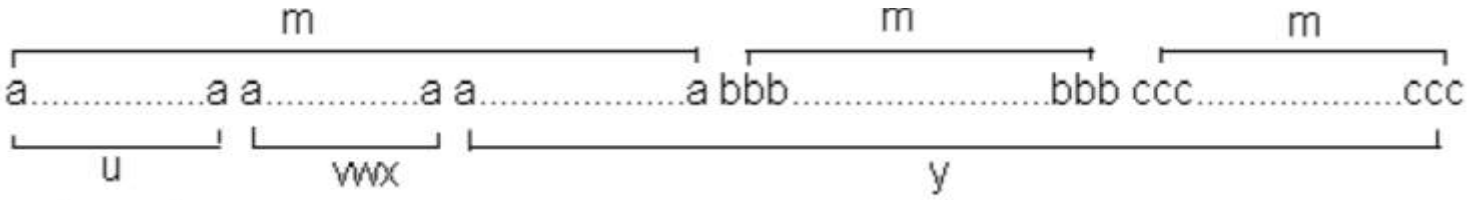
Theorem

- 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.

Application(Example)

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

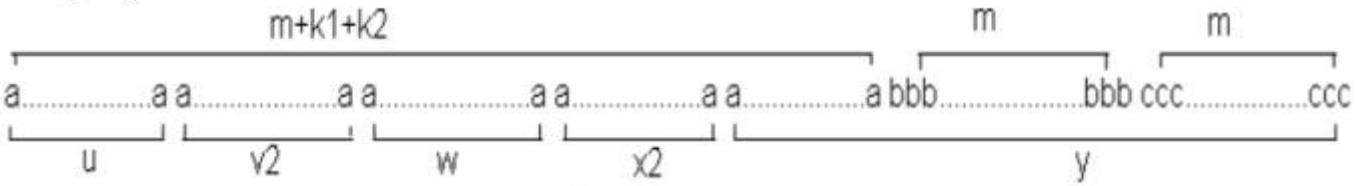
- To prove L is not CFL, we can make use of pumping lemma for CFL. Let L be CFL. Let any string in the language is $a^m b^m c^m$
  - $z = uvwxy$, $|vwx| \leq m$, $|vx| > 0$
- **Case I:**
  - vwx is in a^m



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

Application(Example)

- Now pump v and x then



- But by pumping lemma, $uv^2xy^2z = a^{m+K1+k2}b^mc^m$ does not belong to L as, k1+k2.
- Hence, it shows contradiction that L is CFL. So, L is not CFL.
- Note: other cases can be done similarly.

Closure Properties of CFL

1. **The context free language are closed under union**

- i.e. Given any two context free languages L1 and L2, their union L1 U L2 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.

Closure Properties of CFL

- Let G1 = (V1, T1, P1 and S1) and G2 = (V2, T2, P2 and S2) be two context free grammars defining the languages L(G1) and L(G2). Without loss of generality, let us assume that they have common terminal set T, and disjoint set of nonterminals.
- Because, the non-terminals are distinct so the productions P1 and P2.
- Let S be a new non-terminal not in V1 and V2. Then, construct a new grammar G = (V, T, P, S) where;
  - V = V1 U V2 U {S}
  - P = P1 U P2 U {S→S1, S→ S2}

Closure Properties of CFL

- 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 (G1) U L (G2).
- For this, Suppose that x ε L (G1). Then there is a derivation of x ;
  - S1→*x
- But in G we have production, S→ S1
- So there is a derivation of x also in G as:
  - S→ S1→*x
- Thus, x ε L (G). Therefore, L (G1) ⊆ L (G). A similar argument shows L (G2) ⊆ L (G).

Closure Properties of CFL

- So, we have,
- L (G1) U L (G2) is subset of L (G)
- Conversely, suppose that x ε L (G). Then there is a derivation of x in G as:
- S→ β →* x
- Because of the way in which P is constructed, β must be either S1 or S2.
- Suppose β =S1. Any derivation in G of the form S1→*x must involve only productions of G1 so, S1→*x is a derivation of x in G1.

Closure Properties of CFL

- Hence, $\beta = S1 \rightarrow x \in L(G1)$. A simpler argument proves that $\beta = S1$ implies $x \in L(G1)$.
- Thus L(G) is subset of L(G1) U L(G2).
- It follows that L(G) = L(G1) U L(G2)

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

2. **The CFLs are closed under concatenation**
3. **The CFLs are closed under kleen closure**

Closure Properties of CFL

- context free languages are not closed under some cases like, intersection and complementation.
- **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;
  - $L1 = \{a^n b^n c^i \mid n \geq 1\ i \geq 1\}$
  - $L2 = \{a^i b^n c^n \mid n \geq 1\ i \geq 1\}$

Closure Properties of CFL

- Clearly,
- $L = L1 \cap L2$. to see why, observe that L1 requires that there be the same number of a's + b's, while L2 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