Introduction to Context Free Grammar

The term *grammar* applied to a language refers to the mechanism for constructing phrases and sentences that belong to the language. A grammar consist a set of rules, by which strings in a language can be generated.

Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings.

Context-free grammars are studied in fields of theoretical computer science, compiler design, and linguistics. CFG's are used to describe programming languages and parser programs in compilers can be generated automatically from context-free grammars.

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

Formally, a context free grammar is defined by 4-tuples (V, T, P, S)

where,

V= set of variables or non terminals

T= Set of terminal symbols

P = Set of rules or productions. Each production rule has the form $A \rightarrow \beta$, where $A \in V$ and $\beta \in (V \cup T)^*$

S = Start symbol and $S \in V$.

Thus, CFG consist of a collection of substitution rules, also called productions with each rules being a variable and a string of variables and terminals. Each rule appears as a line in the grammar.

The symbols involved in the production may be variable or terminal symbols. The variable symbols are 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 designate as a start variable S. It usually occurs on the left hand side of the topmost rule. In CFG, the left hand side of rule always consists of a variable and the right hand side consists of variables, terminals or combination of them.

For example,

Rule 1: $S \rightarrow \epsilon$

Rule 2: $S \rightarrow 0S1$

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 terminal symbols defining T; S is a variable symbol defining V; S is start symbol.

In the above example there are two rules in production set. Thus formally, the grammar is defined as $(V=\{S\}, T=\{\epsilon, 0, 1\}, P=\{S \rightarrow \epsilon, S \rightarrow 0S1\}, S=\{S\})$.

The production set can also be written as $P=\{S \rightarrow \epsilon \mid 0S1\}$ where the symbol | is used to separate two productions from the same variable S.

Recursive Structure in Grammars

Consider a CFG representing the language over $\Sigma = \{a, b\}$ which is palindrome language, defined by following productions;

$S \rightarrow \epsilon|a|,b$

$S \rightarrow aSa$

$S \rightarrow bSb$

The above grammar generates the strings like ϵ, a, b, aa, bb, aba, bab, abba, baab, bbbb etc. Each of the strings in palindrome has a structure of wwr, where wr is a reverse of string w. The regular expressions do not have capacity to represent such structure of strings but context free grammars can, as above. A significant reason behind this is that regular expressions are a way of expressing regular grammars which are processed by the state machines like DFA and NFA. Such machines do not have any storage memory associated with it so it's hard to remember the string w while processing wr. In contrast to this CFG represents context free language which is proceed by a machine called push down automaton. It has memory structure to store the history of processed inputs, thus can parse the wr by storing w in its memory and hence can determine the recursive structure easily.

Meaning of context free in CFG

Consider an example;

s→amb

m→A|B

A→∈ |Aa

B→ ∈|bB

Now, consider a string aaAb, which is an intermediate stage in the generation 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 the string aA no matter what the surrounding strings are; in other words, independently of the context of A. However if there is a production of form aaAb→aBb (but not of the form A→B) , the grammar is context sensitive since A, can be replaced by B only when it is surrounded by the strings "aa" and "b". This in contraxt to context free is context sensitive in the grammar.

Example 1: CFG for language 0^n1^n where n≥0.

Solution: S→ ∈

S→ 0S1

Example 2: CFG for algebraic expression involving binary operators + and -, left and right parenthesis, and a single identifier a.

Solution:

S→ S+S

S→ S-S

S→ (S)

S→ a

Example 3: CFG for language L = {aa, ab, ba, bb}

Solution:

S→ AA

A→ a

A→ b

Example 4: CFG for language 1^n where n≥0.

Solution:

S→ 1S

S→∈

Example 5: CFG for (0+1)*

Solution:

S→ 1S

S→ 0S

S→∈

Example 6: CFG for string having atleast length two. i.e (0+1)(0+1)(0+1)*

Solution:

S→AAB

A→ 1 | 0

B→ 1S | 0S | ∈

DERIVATION USING A GRAMMAR RULE

The process of producing strings using the production rules of the grammar is called derivation.

A derivation of a context free grammar is a finite sequence of strings β_0 β_1 β_2.....β_n such that:

- For $0 \le i \le n$, the string $\beta_i \varepsilon (V \cup T)^*$
- $B_0 = S$
- 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 (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) | S-S| S*S| a

Here to generate 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 (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

1. Leftmost derivation
2. Rightmost derivation

Leftmost derivation

A derivation in a context-free grammar is a *leftmost* derivation (LMD) if, at each step, a production is applied to the leftmost variable occurred in the current string. It is denoted by $\underset{lm}{\rightarrow}$.

Consider a grammar defined as

S → S*S

S→ S+S

S→ S-S

S→ S/S

S→ (S)

S→ a

Now the leftmost derivation of the string $a + (a * a)$ is shown as:

$S \underset{lm}{\rightarrow} S+S$ (Rule S→ S+S)

$S \underset{lm}{\rightarrow} a+S$ (Rule S→ a)

$S \underset{lm}{\rightarrow} a+(S)$ (Rule S→ (S))

$S \underset{lm}{\rightarrow} a+(S*S)$ (Rule S→ S*S)

$S \underset{lm}{\rightarrow} a+(a*S)$ (Rule S→ a)

$S \underset{lm}{\rightarrow} a+(a*a)$ (Rule S→ a)

Rightmost derivation

A derivation in a context-free grammar is a *right most* derivation (RMD) if, at each step, a production is applied to the rightmost variable occurred in the current string. It is denoted by $\underset{rm}{\rightarrow}$.

Consider a grammar defined as

S → S*S

S→ S+S

S→ S-S

S→ S/S

S→ (S)

S→ a

Now the rightmost derivation of the string $a + (a * a)$ is shown as:

$S \underset{rm}{\rightarrow} S+S$ (Rule S→ S+S)

$S \underset{rm}{\rightarrow} S+(S)$ (Rule S→ (S))

$S \underset{rm}{\rightarrow} S+(S*S)$ (Rule S→ S*S)

$S \underset{rm}{\rightarrow} S+(S*a)$ (Rule S→ a)

$S \underset{rm}{\rightarrow} S+(a*a)$ (Rule S→ a)

$S \underset{rm}{\rightarrow} a+(a*a)$ (Rule S→ a)

#Consider a Grammar;

$S \rightarrow aAS \mid a$

(A) $S \rightarrow SbA \mid SS \mid ba$

Given a string aaabaaa, give leftmost and rightmost derivation.

Leftmost Derivation:

$S \underset{lm}{\rightarrow} aAS$

$S \underset{lm}{\rightarrow} aSS$; rule $A \rightarrow SS$

$S \underset{lm}{\rightarrow} aaSS$; rule $S \rightarrow a$

$S \underset{lm}{\rightarrow} aaaSS$; rule $S \rightarrow aAS$

$S \underset{lm}{\rightarrow} aabaSS$; rule $A \rightarrow ba$

$S \underset{lm}{\rightarrow} aaabaaS$; rule $S \rightarrow a$

$S \underset{lm}{\rightarrow} aaabaaa$; rule $S \rightarrow a$

Rightmost Derivation:

$S \underset{rm}{\rightarrow} aAS$

$S \underset{rm}{\rightarrow} aAa$; rule $S \rightarrow a$

$S \underset{rm}{\rightarrow} aSSa$; rule $A \rightarrow SS$

$S \underset{rm}{\rightarrow} aSaASa$; rule $S \rightarrow aAS$

$S \underset{rm}{\rightarrow} aSaAaa$; rule $S \rightarrow a$

$S \underset{rm}{\rightarrow} aSabaaa$; rule $A \rightarrow ba$

$S \underset{rm}{\rightarrow} aaabaaa$; rule $S \rightarrow a$

Given CGF:

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA \mid \epsilon$

$B \rightarrow b \mid bS \mid aBB$

Given string babaababbaa, shows leftmost and rightmost derivation.

Leftmost derivation:

$S \underset{lm}{\rightarrow} bA$

$S \underset{lm}{\rightarrow} baS$; rule $A \rightarrow aS$

$S \underset{lm}{\rightarrow} babA$; rule $S \rightarrow bA$

$S \underset{lm}{\rightarrow} babbAAS$; rule $A \rightarrow bAA$

$S \underset{lm}{\rightarrow} babbaAS$; rule $A \rightarrow a$

$S \underset{lm}{\rightarrow} babbaaS$; rule $A \rightarrow aS$

$S \underset{lm}{\rightarrow} babbaabA$; rule $A \rightarrow bA$

$S \underset{lm}{\rightarrow} babbaabaS$; rule $A \rightarrow aS$

$S \underset{lm}{\rightarrow} babbaababA$; rule $A \rightarrow bA$

$S \underset{lm}{\rightarrow} babbaababbAA$; rule $A \rightarrow bAA$

$S \underset{lm}{\rightarrow} babbaababbaA$; rule $A \rightarrow a$

$S \underset{lm}{\rightarrow} babbaababbaa$; rule $A \rightarrow a$

Rightmost Derivation:

$S \underset{rm}{\rightarrow} bA$

$S \underset{rm}{\rightarrow} baS$; rule $A \rightarrow a\,S$

$S \underset{rm}{\rightarrow} babA$; rule $S \rightarrow bA$

$S \underset{rm}{\rightarrow} babbAA$; rule $A \rightarrow bAA$

$S \underset{rm}{\rightarrow} babbAa$; rule $A \rightarrow a$

$S \underset{rm}{\rightarrow} babbaSa$; rule $A \rightarrow a\,S$

$S \underset{rm}{\rightarrow} babbaaBa$; rule $S \rightarrow a\,B$

$S \underset{rm}{\rightarrow} babbaabSa$; rule $B \rightarrow b\,S$

$S \underset{rm}{\rightarrow} babbaabaBa$; rule $S \rightarrow a\,B$

$S \underset{rm}{\rightarrow} babbaababSa$; rule $B \rightarrow b\,S$

$S \underset{rm}{\rightarrow} babbaababbAas$; rule $S \rightarrow b\,A$

$S \underset{rm}{\rightarrow} babbaababbaa$; rule $A \rightarrow a$

Direct derivation

During the derivation of a string from given productions a grammar, if there is a production $\alpha_1 \Rightarrow \alpha 2$

then α_2 can be derived directly from α_1, hence it is direct derivation.

Otherwise If α_2 can be derived from α_1 with zero or more steps of the derivation, then it is derivation and is represented as $\alpha_1 \overset{*}{\Rightarrow} \alpha_2$

For example; S→asa | a b| |a |b| |∈|

Direct derivation;- S⇒ab

S⇒aSa

⇒aasaa

⇒aaabaa

Thus, s$\overset{*}{\Rightarrow}$ aaabaa is just a derivation.

Language of CFG

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 ε T* | S→* x}

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

Parse Tree /Derivation Tree

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:

- Root node which is labeled by the start symbol.
- Interior nodes each of which in the parse tree is variables.
- Leaf node each of which in the parse is labeled by a terminal symbol or ∈.

If an interior node is labeled with a non terminal A and its childrens are x_1, x_2 .....x_n from left to right there is a production P as:

A→ $x_1, x_2, \ldots\ldots\ldots\ldots x_n$ for each x_i ε T.

Consider the grammar

1. S → S*S
2. S→ S+S
3. S→ S-S
4. S→ S/S
5. S→ (S)
6. S→ a

The parse tree for the string *a + (a *a)*is shown as follows:

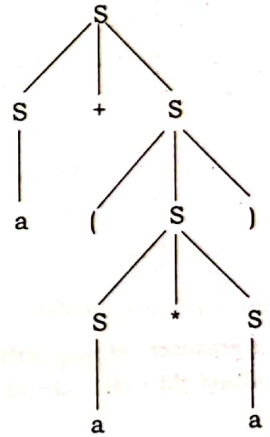


Fig: Derivation tree for *a + (a *a)*

Consider the Grammar

1. S→I
2. S→S+S
3. S→S * S
4. S→(S)
5. I→a
6. I→b
7. I→Ia

8. I→Ib
9. I→I0
10. I→I1

Construct the parse tree for $a * (a+b00)$.

The parse tree is;

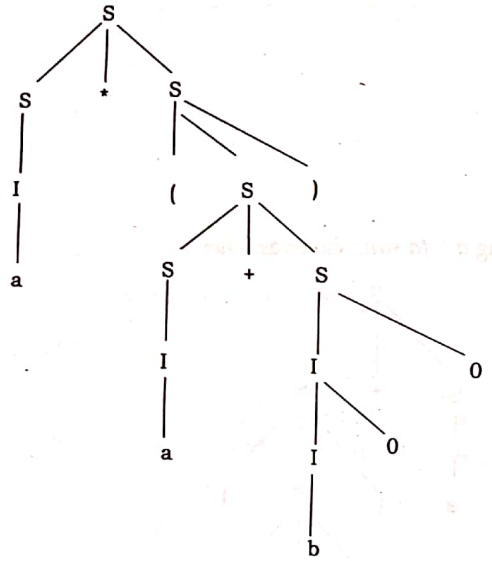


Fig: Parse tree for a * (a+b00)

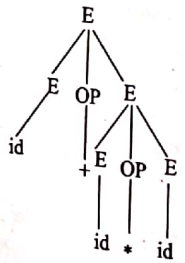
Example 7: Construct a grammer defining arithmetic expression and generate parse tree for id + id *id and (id + id) * (id + id).

Solution:

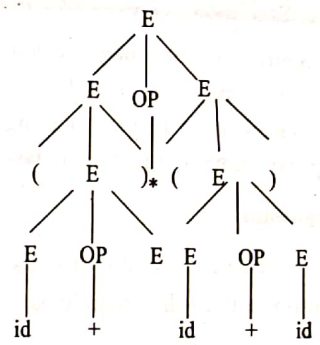
E → E OPE | (E) | id

OP → + | - | * | / | ↑

Parse tree for id + id * id



Parse tree for (id + id) * (id + id)



Ambiguity in a CFG

A grammar G = (v_1, t_1 p_1 s) is said to be ambiguity if there is a string w ∈ 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.

Example 8: Consider the grammar

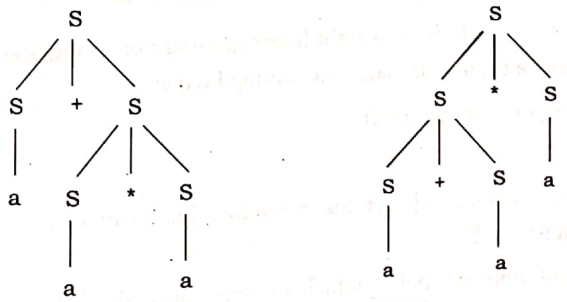
S → a | S + S | S * S | (S)

The string a + a * a has the two leftmost derivations

S → S + S → a + S → a + S * S → a + a * S → a + a * a

S → S * S → S + S * S → a + S * S → a + a * S → a + a * a

which correspond to the derivation trees



Hence the above grammar is ambiguous grammar.

Inherently ambiguity

A context free language L is said to be inherently ambiguous if all its grammars are ambiguous. E.g. L= {$0^i1^j2^k$ | i=j or j=k}.

REGULAR GRAMMAR

A regular grammar represents a language which is represented by regular expressions. The regular grammar is accepted by NFA and DFA and the language of the grammar is called regular language. A regular grammar is a subset of CFG. The regular grammar may be may be either left or right linear.

Right Linear Regular Grammar

A regular grammar is which all of the productions are of the form $A \rightarrow wB$ or $A \rightarrow w$ where $A,B \in V$ and $w \in T^*$ is called right linear.

For Example;

$S \rightarrow 00B \mid 11C$

$B \rightarrow 11C \mid S \mid 1$

$C \rightarrow 00B \mid 00$

Left Linear Regular Grammar

A grammar is which all of the production are of the form $A \rightarrow Bw$ or $A \rightarrow w$, where $A, B \in V$ and $w \in T^*$ is called left linear.

For example

$S \rightarrow B00 \mid C11$

$B \rightarrow C11 \mid S \mid 1$

$C \rightarrow B00 \mid 00$

Equivalence of Regular Grammar and Finite Automata

1. Let $G = (V, T, P, S)$ 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 $[\alpha]$ 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 $\sum$ which are terminal symbols of G.

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

$\{[\epsilon]\}$ – is the set of final states in finite automata.

δ – is a transition function and is defined as:

If A is a variable then,

$\delta([A], \epsilon) = [\alpha]$ such that $A \rightarrow \alpha$ is a production.

If 'a' is a terminal and α is in $(T \cup V)^*$ then

$\delta([a\alpha], a) = [\alpha]$

Example 9:

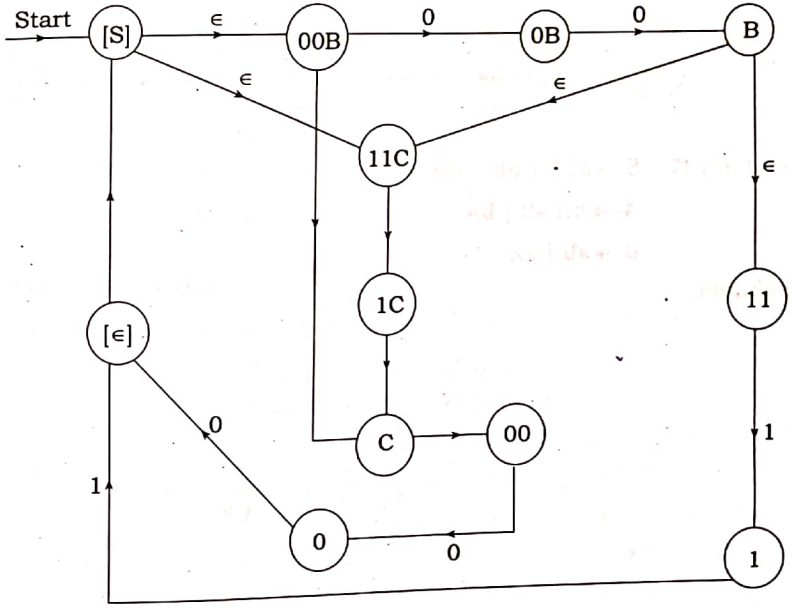
$S \rightarrow 00B \mid 11C \mid \epsilon$

$B \rightarrow 11C \mid 11$

$C \rightarrow 00B \mid 00$

Solution:

The finite automaton for the above grammar is given as;



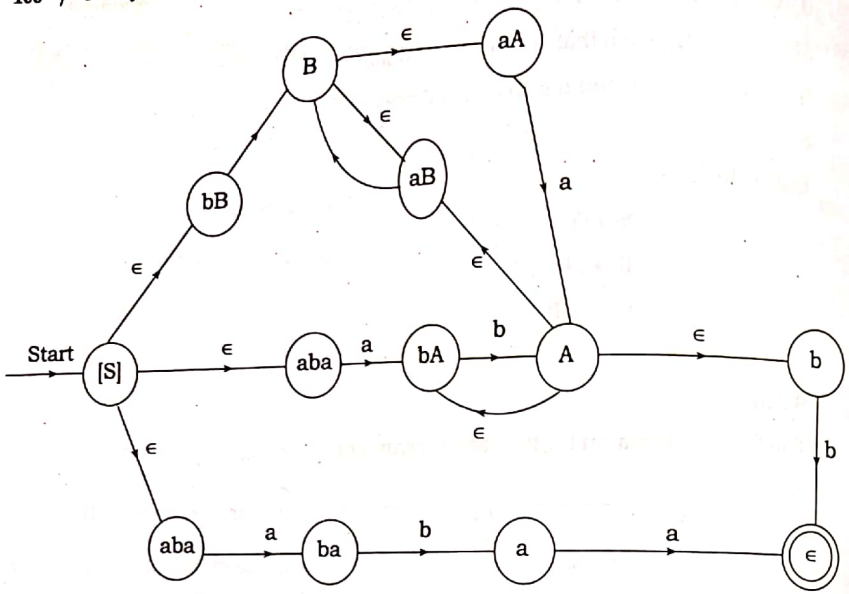
Example 10: $S \rightarrow abA \mid bB \mid aba$

$A \rightarrow b \mid aB \mid bA$

$B \rightarrow aB \mid aA$

Solution:

Here ϵ does not appear on the body part of any productions, but we introduce state [ϵ] and select it as accepting state. Thus the finite automaton for the above grammar is given as:

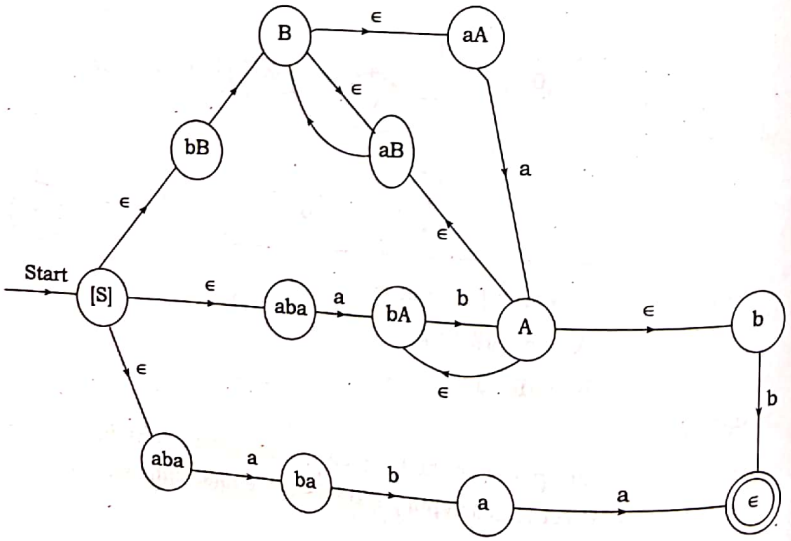


Example 11: S → abA | bB | aba

A → b | aB | bA

B → aB | aA

Solution:

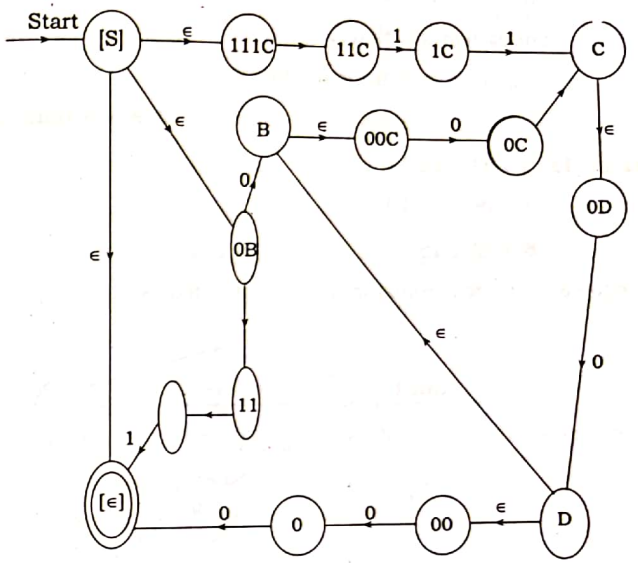


Example 12: S → 0B | 111C | ϵ

B → 00C | 11

C → 00B | 0D

D → 0B | 00



Another Approach

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 ϵ as S → *ϵ, S being start symbol of grammar, then start state of the automaton will also be final state.

d. The transition function for the automaton is defined as;

- For each production A→ aB

  $\delta(A, a) = B.0$

  So there is an arc from state A to B labeled with a.

- For each production A→ a,

  $\delta(A, a)$ = final state of automaton

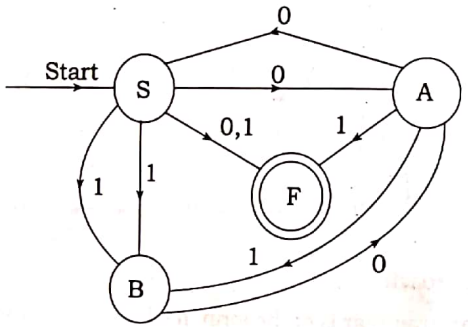
- For each production A→ ∈, make the state A as finite state.

Example 13: S→ 0A | 1B | 0 | 1

A→ 0S | 1B | 1

B→ 0A | 1S

The equivalent finite automaton can be configured as;



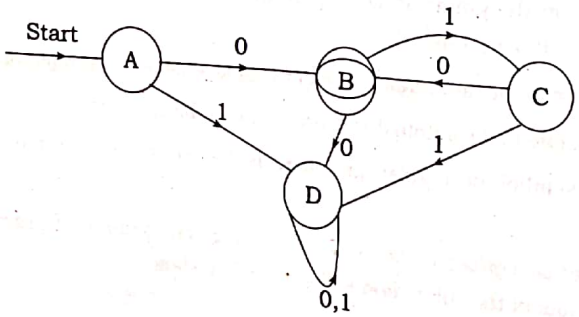
Example 14: A→ 0B | 1D | 0

B→ 0D | 1C | ∈

C→ 0B | 1D | 0

D→ 0D | 1D

The equivalent finite automaton can be configured as;



SIMPLIFICATION OF CFG

In a CFG, it may not be necessary to use all the symbols in V and T, or all the productions in P for deriving strings from the grammar. Thus simplification of CFG refers to eliminate those symbols and productions in G which are not useful or are unnecessary for the derivation of strings. The Simplification of CFG involves following;

i. Eliminating ∈-productions.

ii. Eliminating unit productions

iii. Eliminating useless symbols

Eliminating ∈-productions (Eliminating Nullable variables)

To eliminate ∈-productions we have to find nullable variables. Nullable variables are the variables that produce ∈.

Recursive Definition of Nullable variables

- Every variable A for which there is a production A →∈ is nullable.
- If $A_1, A_2, \ldots, A_k$ are nullable variables (not necessarily distinct), and B → $A_1A_2 \ldots A_k$ then B is also nullable variable.

Algorithm for elimination of ∈-productions

1. Find all the nullable variables.
2. For each production **A** → $A_1A_2 \ldots A_k$, construct all productions **A** → **X**where **X**is obtained from '$A_1A_2 \ldots A_k$' by removing zero, one or multiple nullable variables found in step 1.
3. Combine the original productions with the result of step 2 and remove**∈-productions**.

Example 15: Consider the grammar:

S→AB

A→aAA | ∈

B→bBB | ∈

Here,

A→∈	A is nullable
B→∈	B is nullable
S→AB	Since A and B are both nullable hence S is nullable.

Now, Removing of ∈ -production we get the grammar:

S→AB|A|B

A→aAA|aA|a

B→bBB|bB|b

Example 16: Consider the grammar:

S→ ABC

A→ BB | ∈

B→ CC | a

C→ AA | b

Here,

A→∈ A is nullable.

C→ AA$\overset{*}{\rightarrow}$ ∈ C is nullable

B→ CC$\overset{*}{\rightarrow}$ ∈ B is nullable

S→ ABC$\overset{*}{\rightarrow}$ ∈ S is nullable

Now for removal of ∈ -production;

S→ABC | AB | BC | AC | A | B | C

A→BB | B

B→CC | C | a

C→ AA | A | b

Example 17: Consider the grammar

S→ AACD

A→ aAB | ∈

C→ aC | a

D→ aDa | bDb | ∈

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

S→ AACD | ACD | AAC | AC | C

A→ aAb | ab

C→ aC | a

D→aDa |abDb | bb

Eliminating Unit Production

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

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$\overset{*}{\rightarrow}$ 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 also a unit pair.

Now, to eliminate unit pair (A , B) i.e. unit production A→ B simply replace B by its body part and then remove all the unit productions.

Example 18: Consider the Grammar

S→ S + T | T

T→ T* F | F

F→ (S) | a

Remove the unit production

Solution:

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 removing unit production we get:

S→ S + T | T * F | (S) | a

T→ T * F | (S) | a

F→ (S) | a

Example 19: Simply the grammar G = (V, T, P, S) defined by following productions.

S→ ASB | ∈

A→ aSA | a

B→ SbS | A | bb | ∈

Solution:

Removing ∈-productions;

Here, S→ ∈ and B→ ∈ So S and B are nullable.

Now, removing ∈-productions, it yields;

S→ ASB | AS | AB | A

A→ aSA | aA

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→ aSA | aA

B→ SbS | Sb | bS | b | bb | aAS | aA

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

Eliminate Useless Symbols

Useless symbols are variables or terminals that do not appear in any derivation of a terminal string from the start symbol.

Formally, we say a symbol X is useful for a grammar G = (V, T, P, S) if there is some derivation of the form $S \overset{*}{\rightarrow} \alpha X \beta \overset{*}{\rightarrow} w$, where w is in T*. Here, X may be either variable or terminal and the sentential form αXβ might be the first or last in the derivation. If X is not useful, we say it is useless.

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

- We say X is generating if $X \overset{*}{\rightarrow} w$ 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 $S \overset{*}{\rightarrow} \alpha X \beta$ for some α and β.

Thus if we eliminate the non generating symbols first 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, Identify non-reachable symbols and eliminating those productions which contain non-reachable symbols.

Example 20: Consider a grammar defined by following productions:

S→ aB | bX

A→ Bad | bSX | a

B→ aSB | bBX

X→ SBd | aBX | ad

Eliminate the useless symbols.

Solution:

Here, A and X are generating symbols, since they produce strings composed of terminals. i.e

A→ a

X→ ad

Also, S→ bX and X generates terminal string so S can also generate terminal string. Hence, S is also generating symbol.

But B cannot produce any terminal string, 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 any derivation of the form $S \overset{*}{\rightarrow} \alpha A \beta$ in the grammar. i.e. A cannot be reached from the starting variable S.

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

S→ bX

X→ ad

This is the grammar with only useful symbols.

Example 21: Consider a grammar defined by following productions:

S → AB | CA

B→ BC | AB

A → a

C→ aB | b

Eliminate useless symbols from the above grammar.

Solution:

Here A and C are generating symbols, since they produce a, b respectively.

Also, Since S→ CA and C and A are generating symbol hence S is also generating symbol.

But B is non-generating symbol, since it cannot produce any terminal string.

Hence, the new grammar after removing non-generating symbols is as;

$S \rightarrow CA$

$A \rightarrow a$

$C \rightarrow b$

Here, all symbols reachable. Hence the grammar having only useful symbols is given as follows:

$S \rightarrow CA$

$A \rightarrow a$

$C \rightarrow b$

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;

$A \rightarrow BC$ and

$A \rightarrow a$ where A, B, C ε V and a ε T

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

- ϵ-production
- Unit production
- Useless symbols

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

Proof: Let G be the grammar that represents CFL without ϵ-production. Now we can simplify this grammar into its equivalent grammar that does not contain unit productions and useless symbols. Thus its equivalent grammar is of the form:

i. $A \rightarrow a$

ii. $A \rightarrow X_1X_2X_3.....X_k$

Here we do not have to change the production (i), since it is already in CNF form.

For production (ii) if k = 2 and it contains some terminal symbol in body part then just replace this terminal symbol by introducing new variable and add a new production having head part as new variable and body part as terminal symbol, which converts it into CNF form.

For production (ii) if k > 2 and it contains some terminal symbols in body part then first just replace these each terminal symbols by introducing new variables and add a new productions having head part as new variable and body part as terminal symbol. Let the production then changes into the form:

$A \rightarrow B_1B_2B_3.....B_k$

Now we break this production into group of productions with two variables in each body. For this we introduce k-2 new variables, C_1, C_2, ..., C_{k-2}. The original production is then replaced by the k-1 new productions as:

$A \rightarrow B_1C_1$, $C_1 \rightarrow B_2C_2$, $C_2 \rightarrow B_3C_3$, ..., $C_{k-2} \rightarrow B_{k-1}B_k$,

Which is in the CNF form and represents (CFL) without ϵ-production

Example 22: Consider Grammar

$S \rightarrow AAC$

$A \rightarrow aAb \mid \epsilon$

$C \rightarrow aC \mid a$

Convert it into CNF form.

Solution:

1) Now, removing ϵ- productions;

Here, A is nullable symbol as $A \rightarrow \epsilon$

So, eliminating such ϵ-productions, we have;

$S \rightarrow AAC \mid AC \mid C$

$A \rightarrow aAb \mid ab$

$C \rightarrow aC \mid a$

2) Removing unit-productions;

Here, the unit pair we have is (S, C) as $S \rightarrow C$

So, removing unit-production, we have;

$S \rightarrow AAC \mid AC \mid C$

$A \rightarrow aAb \mid ab$

$C \rightarrow aC \mid a$

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;

$S \rightarrow AC_2 \mid AC \mid C_1C \mid a$

$A \rightarrow C_1C_3 \mid C_1b_1$

$C_1 \rightarrow a$

$B_1 \rightarrow b$

$C_2 \rightarrow AC$

$C_3 \rightarrow AB_1$

$C \rightarrow C_1C \mid a$

Example 23: Simplify following grammars and convert to CNF;

$S \rightarrow ASB \mid \epsilon$

$A \rightarrow aAS \mid a$

$B \rightarrow SbS \mid A \mid bb$

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;

$S \rightarrow AC_2 \mid AB$

$A \rightarrow C_1C_3 \mid C_1A \mid a$

$B \rightarrow SB_2 \mid SB_1 \mid B_1 \mid C_1C_3 \mid C_1A \mid B_1B_1 \mid b$

$C_1 \rightarrow a$, $C_2 \rightarrow SB$, $B_1 \rightarrow b$, $C_3 \rightarrow AS$, $B_2 \rightarrow B_1S$

Example 24: Simplify the grammar and convert to CNF:

$S \rightarrow aaaaS \mid aaaa$

Solution:

Here, we do not have any ϵ-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;

$S \rightarrow AA_1 \mid AC$

$A_1 \rightarrow AA_2$

$A_2 \rightarrow AA_3$

$A_3 \rightarrow AS$

$C \rightarrow AC_1$

$C_1 \rightarrow AA$

$A \rightarrow a$

LEFT RECURSIVE GRAMMAR

In a context-free grammar **G**, if there is a production in the form $A \overset{*}{\rightarrow} A\alpha$ where **A** is a non-terminal and 'α' is a string composed from (V U T)*, it is called a left **recursive production**. The grammar having a left recursive production is called a **left recursive grammar.**

For example;

$S \rightarrow AA \mid o$

$A \rightarrow AAAS(\alpha_1) \mid os\ (\beta_1) \mid 1(\beta_2 \mid$

Removal of Left Recursion

Let $A \rightarrow A\alpha \mid \beta$

Here β does not start with A. Then, the left-recursive pair of production could be replaced by the non-recursive production as;

$A \rightarrow \beta A^1$

$A^1 \rightarrow \alpha A^1 \mid \in$; without changing the set of strings derivable from A.

No matter how many A-productions there are, we can eliminate immediate left recursion from then.

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 β_i does not start with A.

Then we can resolve left recursive as;

$A \rightarrow \beta, A^1 \mid \beta_2 a^1 \mid \ldots\ldots \mid \beta_N a^1$

$a^1 \rightarrow \alpha_1 a^1 \mid \alpha_2 a^1 \mid \ldots\ldots \mid \alpha_M a^1 \mid \in$

Equivalently, these productions can be rewritten as;

$A \rightarrow \beta_1 A^1 \mid \beta_2 A^1 \mid \ldots\ldots\ldots \mid \beta_n A^1 \mid \beta_1 \mid \beta_2 \ldots\ldots \mid \beta_m$

$A^1 \rightarrow \alpha_1 A^1 \mid \alpha_2 A^1 \mid \ldots\ldots \mid \alpha_n A^1 \mid \alpha_1 \mid \alpha_2 \mid \ldots\ldots\ldots \mid \alpha_m$

Example 25: Remove left recursive production from the following grammar:

$S \rightarrow AA \mid 0$

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

Solution:

Here, the production $A \rightarrow AAS$ is immediate left recursive. Now comparing $A \rightarrow AAS \mid 0S \mid 1$ with (1) above we get,

$\alpha_1 = AS$

$\beta_1 = 0S$

$\beta_2 = 1$

So removing left recursion we have,

$S \rightarrow AA \mid 0$

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

$A' \rightarrow ASA' \mid AS$

Example 26: Remove left recursive production from the following grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid a$

Solution:

Here, $E \rightarrow E+T$ and $T \rightarrow T*F$ are left recursive so removing the left recursive

$E \rightarrow TA' \mid T$

$A' \rightarrow +TA' \mid +T$

$T \rightarrow FB' \mid F$

$B' \rightarrow *FB' \mid *F$

$F \rightarrow (E) \mid a$

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. $\alpha \in V^*$

we can rewrite as;

$A \rightarrow av^*$, with $a \in t$

$A \rightarrow av^+$

$A \rightarrow a$; with $a \in t$.

This form, called Greibach Normal form, other sheila Greibach, who first gave a way to construct such grammars. converting, 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 free 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.....

Example 27: Convert the following grammar into GNF

$S \rightarrow AA \mid 0$

$A \rightarrow SS \mid 1$

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 =β_1 and 1of the form=β_2)*

Now, removing the immediate left recursion;

$S \rightarrow AA \mid 0$

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

$A' \rightarrow ASA' \mid AS$

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

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

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

$A' \rightarrow ASA' \mid AS$

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

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

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

$A' \rightarrow 0SA'SA' \mid 1A'SA' \mid 0SSA' \mid 1SA' \mid 0SA'S \mid 1A'S \mid 0SS \mid 1$

Bakus- Naur form

This is another notation used to specify the CFG .It is named so, after Jhon Bakus, who invented it, and peter Naur, who refined it. The Bajus-Naur form is used to specify the syntactic rule of many computer languages, including Java. Here, concept is similar to CFG, only the difference instead of using symbol (→) in a production, we use symbol (::=). We enclose all non terminal in only brackets, < >.

Thus A BNF grammar is a set of derivation rules, written as <symbol> ::= __expression__

Where <symbol> is a nonterminal, and the __expression__ consists of one or more sequences of symbols; more sequences are separated by the vertical bar "|", indicating a choice, the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are terminals. On the other hand, symbols that appear on a left side are non-terminals and are always enclosed between the pair <>.

The "::=" means that the symbol on the left must be replaced with the expression on the right.

For example,

<integer>::=<digit> | <integer><digit>

<digit>::= 0 | 1 | 2 | ..... | 9

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 (CSG) is a formal grammar in which left hand sides and right hand sides of any production may be surrounded by a context of terminal and non-terminal symbols.

Formally, CSG can be defined as;

A formal grammar, G = (V, T, p, S) is context sensitive if all the production 'P' are of the form;

$\alpha A\beta \rightarrow \alpha\gamma\beta$, where $A \in V$, $\alpha \in V$, $\alpha\beta \in (v\ u\ t)^*$ and $\gamma \in (v\ u\ t)^+$.

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. Thus the production $A \rightarrow \gamma$ can only be applied in contexts where α occurs to left of A and β occurs to right of A.

Example 28: $\{a^n b^n c^n \mid n \geq 1\}$ is a context sensitive language dfined as;

$S \rightarrow aSBC \mid aBC$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bc \rightarrow bc$

$cC \rightarrow cc$

Pumping Lemma for Context Free Languages

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.

Theorem: Suppose L is a context-free language. Then there exist an integer n such that for every string $z \in L$ with $|z| \geq n$, zcan be written as$z = uvwxy$, for some strings $u, v, w, x,$ and y satisfying

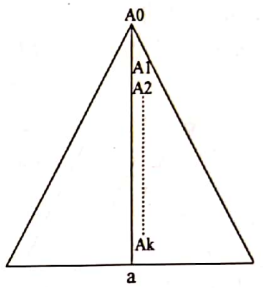
1. $|vx| > 0$
2. $|vwx| \leq n$
3. for every $i \geq 0$, $uv^iwx^iy \in L$

It means that the string can be divided into five parts so that the second and fourth parts may be repeated together any number of times and the resulting string still remains in the language.

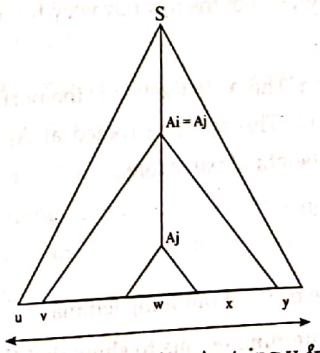
Proof: First we can find a CNF grammar for the grammar G of the CFL L, that generates L-{∈}. Choose $n = 2^m$, Next suppose z in L is of length at least n i.e. $|z| \geq b$.

any parse tree for Z must have high m +1, the if it would be less that will, i.e. say m then by the lemma for size of parse tree. $|z| = 2^{m-1} = \frac{2^m}{2} = \frac{n}{2}2$ is contracting so it should be m+.

Let us consider a path of maximum length is tree for z. as show below, where k is at least m & path is of length k.



Since, $K \geq m$, these are at least (k +1) or (M +1) occurrences of variables $A_0 \ldots\ldots A_k$ on the path. But there are only in variable in grammar, so at least two of the last m+1 variable on the path must be same, (be pigeonhole principle). Suppose $A_i = A_j$, then it is possible to divide tree

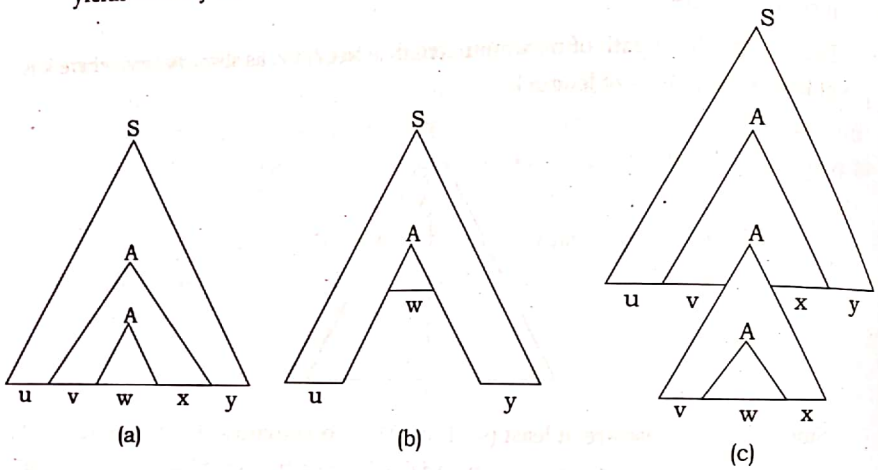


String w is the yield of sub tree rooted at A_j string v & x are to left & right of w in yield of longer sub tree rooted at A_i. If u & y represent beginning & end z. i.e. to right & left of sub tree rooted at A_i then z -uvwxy.

Since, there are not unit productions so v & x both could not be ∈ empty. Means that A_i has always two child corresponding to variables. If we let B denoted the one is not ancestor f A_j, then since w is derived from B does not overlap x . It

follows that either v or x is not empty so , Now, we know $A_i = A_j = A$ say as we found an y two variables in the tree are same.

Them, we can construct new parse tree where we can replace sub tree rooted at A_i, 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 labelled A. The resulting tree yields uv^0wx^0y as;



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

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

Now, to show $|vwx| \leq n$. The A_i in the tree is the portion of o path of path which is root with height m+1. This sub tree rooted at A_1 also have height less than m+1. So by lemma for height of parse tree,

$|v\,w\,x| \leq 2^{m+1-1} = 2^m = n.$

$\therefore |v\,w\,x| \leq n.$

Thus this completes the proof of pumping lemma.

Example 29: Use the pumping lemma to show that the language $L = \{a^m b^m c^m\}$ is not context free.

Solution:

Suppose L is CFL.Then there exists a constant n satisfying the conditions of the pumping lemma. Let $z = a^n b^n c^n$. Clearly z is a member of L and of length atleast n. Using pumping lemma we can break z = uvwxy so that $|vx| > 0$ and $|vwx| \leq n$.

In this case either string vwx is composed from only one alphabet symbol or may be composed from both a's and b's or from both b's and c's but not from all three alphabet symbols.

Case I: If vwx is composed from only one alphabet symbol.

In this case the string uv^2wx^2y cannot contain equal number of a's, b's and c's. Therefore it cannot be the member of L. i.e. $uv^2wx^2y \notin L$.

Case II : If vwx is composed from two alphabet symbols.

i) Substring vwx does not contain any c.

- Consider the string $uv^2wx^2y = uvvwxxy$. Since $|vx| \geq 1$, this string contains more than n many a's or more than n many b's. Since it contains exactly n many c's, it follows that this string is not in the language L. Thisis a contradiction because, by the pumping lemma, the string uv^2wx^2y is inL.

ii) Substring vwx does not contain any a.

- Consider the string $uv^2wx^2y = uvvwxxy$. Since $|vx| \geq 1$, this string contains more than n many b's or more than n many c's. Since it contains exactly n many a's, it follows that this string is not in the language L. This is a contradiction because, by the pumping lemma, the string uv^2wx^2y is inL.

Thus, in all of the two cases, we have obtained a contradiction. Therefore, we have shown that the language A is not context-free.

Example 30: Prove that $L = \{ww : w \in \{a, b\}^*\}$ is not CFL.

Proof:

Suppose L is a context-free language. Then there exists a constant n satisfying the conditions of the pumping lemma. Let $z = a^n b^n a^n b^n$. Clearly z is a member of L and of length $4n \geq n$. Using pumping lemma we can break z = uvwxy so that $|vx| > 0$ and $|vwx| \leq n$.

Based on the location of vwx in the string z, we distinguish three cases:

Case I : The substring vwx overlaps both the leftmost half and the rightmost half of z.

Since $|vwx| \leq n$, the substring vwx is contained in the "middle" part of s, i.e., vxy is contained in the block $b^n a^n$. Consider the string $uv^0wx^0y = uwy$. Since $|vx| \geq 1$, we know that at least one of v and xis non-empty.

- If $v \neq \epsilon$, then v contains at least one b from the leftmost block of b's in z, whereas x does not contain any b from the rightmost block of b's in x. Therefore, in the string uwy, the leftmost block of b's contains fewer b's than the rightmost block of b's. Hence, the string uwy is not contained in L.
- If $y \neq \epsilon$, then y contains at least one a from the rightmost block of a's in z, whereas v does not contain any a from the leftmost block of a's in z. Therefore, in the string uwy, the leftmost block of a's contains more a's than the rightmost block of a's. Hence, the string uwy is not contained in L

Case II: The substring vwx is in the leftmost half of z. In this case, none of the strings uwy, uv^2wx^2y, uv^3wx^3y, uv^4wx^4y, etc., is contained in L. But, by the pumping lemma, each of these strings is contained in L.

Case III: The substring vwx is in the rightmost half of z. This case is symmetric to Case II: None of the strings uwy, uv^2wx^2y, uv^3wx^3y, uv^4wx^4y, etc., is contained in L. But, by the pumping lemma, each of these strings is contained in L.

To summarize, in each of the three cases, we have obtained a contradiction. Therefore, the language L is not context-free.

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 us also context free. These theorem \ lemmas are often called closure properties of context free language, since they show whether or not the class if context free languages us closed under the operation mentioned. Closure properties express the idea that one (or several) languages are context free. Then certain related languages one also context free or not.

Here are some of the principal closure properties for context free languages;

(i) The context free language are closed under union. i.e. given any two context free languages L_1 & L_2, there union $L_1 \cup L_2$ is also context free language.

Proof:

The intuitive idea of the proof is to build a n grammar form the original two, and from start symbol of the new grammar have production to the start symbols of the original two framers.

let $G_1 = (v_1, (T), p_{11}, S_1)$ and $G_2 = (V_2(T), p_2, S_2)^b$ be two context tree grammar defining the language, L(G) & $L(G_2)$ .Without loss of generality, let us assume that they have common terminal set T, and disjoint set non-terminals i.e. W∩1. Because, the terminals are distinct so the production p_1 & p_2.

Let S be a new non-terminal net in v_1 & v_2. Then, construct a new grammar G = (v_1 T_1 P_1 S) where,

$V = V_1 UV_2 U \{S\}$

$P = P_1 U p_2 U \{S \rightarrow S_1, S \rightarrow S_2\}$

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

Supposed that $x \in L(G_1)$. Then these is a derivation of x is $S_1 \stackrel{*}{\Rightarrow} x_1$

But in G we have production,

$S \rightarrow S_1$

so there is a derivation of x above in G as;

$S \Rightarrow S_1 \stackrel{*}{\Rightarrow} x$

Thus, $x \in L(G)$. Therefore, $L(G_1) \leq L(G)$.

A similar argument shows $L(G_2) \leq L(G)$.

So, we have, $L(G_1) \cup L(G_2) \leq L(G)$.

Conversely, suppose that $x \in L(G)$. Then, there is a derivation x is G as;

$S \Rightarrow \beta \stackrel{*}{\Rightarrow} x$.

Because of the way in which p is constructed, β must either S_1 or S_2 suppose $\beta = S_1$. Any derivation in G of the form $S \stackrel{*}{\Rightarrow} x$ must involve only production of G_1 so, $S_1 \stackrel{*}{\Rightarrow} x$ is a derivation of c in G_1.

$\therefore \beta = s_1 \Rightarrow x \in L(G_1)$. A similar argument prove that β = implies x $L(G_2)$. Thus, $L(G) \leq L(G_1) \cup LC$. It flows that $L(G) = L(G_1) U L(G_2)$.

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

2) The CLF_s are closed under Kleen closure (star)

3) The CFL_s are closed under concatenation.

4) The CFL_s are closed under Kleen closure (star).

However, context free languages are not closed under some case live, intersection & complementation.

5) The context free language are not closed under intersection. TO prove this, the idea is to exhibit two CFL_s whose intersection result in the language which in not CFL. For this as we learned in pumping lemma that.

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

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

A grammar for L_1 is;

$S \rightarrow AC$

$A \rightarrow aAb \mid ab$

$C \rightarrow cC \mid c$

In this grammar, A generates all strings of the form a^n b^n & C. Generates all strings of C's.

A grammar for L_2 is is;

$S \rightarrow AB$

$A \rightarrow aA \mid a$

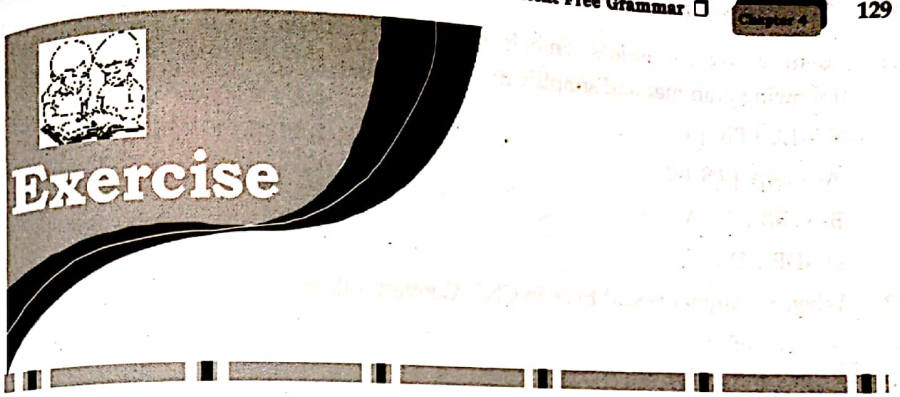
$B \rightarrow bBc \mid bc$. In this grammar, A generates any strings and B generates strings of the form $b^n c^n$.

Clearly, $L = L_1 \cap L_2$. To see why, observe that L_2 required that three be the same number of a's & c's to be equal. A strings in both must have equal number of all three symbols & thus to be in L.

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

6) The CLF's are not closed under complementation, and then they would also be closed under the operation of intersection, contradiction above property. Let L_1 & L_2 be as defined in property then intersection of two languages can be expressed in there of complements as;

$L_1 \cap L_2 = (\overline{\overline{L_1} \cup \overline{L_2}}) = \Sigma^* - L_1) \cup (\Sigma^* - L_2))$ but, we know CFL's are closed under union. So if CFL were close under complementation, then they would be also closed under intersection. Hence, it is not.



1. In Context Free Grammar, explore the meaning of the term "Context Free" with a suitable example.
2. Write a Context Free Grammar representing the "regular expressions".
3. Write a CFG representing strings that start and end with the same symbol over {0, 1}.
4. Write a CFG for a postfix expression over arithmetic operators.
5. Write Context Free Grammar for following
   a. $a^n b^n$
   b. $a^n b^{n+1}$
   c. ww^R
6. Write a Context Free Grammar that defines regular expression.
7. Write a CFG defining *if statement*.
8. What does ambiguity in grammar means? Does ambiguity exists in following grammar?

   S -> SS | aX | bY

   X -> aXX | bS | b

   Y -> bYY | aS | a
9. Write a CFG representing strings that start and end with the same symbol over 0,1
10. A CFG describing strings of letters with the word "main" somewhere in the string.

11. Identify possible useless symbols, epsilon productions and unit productions following grammar and simplify it;

    S→ 1A | 0B | ϵ

    A→ 1AA | 0S | 0

    B→ 0BB | 1 | A

    D→DB | DD

12. When a grammar is said to be in CNF. Convert following grammar to CNF;

    S→ 1A | 0B | ϵ

    A→ 1AA | 0S | 0

    B→ 0BB | 1 | A

    C→CA | CS

13. Simplify following grammars and convert to the CNF:

    a. S→ bA | aB

       A→ bAA | aS | a

       B→ aBB | bS | a

    b. S→ abSb | a | aAb

       A→ bS | aAAb

    c. S→ 1A | 0B

       A→ 1AA | 0S | 0

       A→ 0BB | 1

14. What is left recursive grammar? How removal of left recursion from such grammar is possible.

15. What do you mean by right linear grammar? Give an example of such grammar

16. Given following grammar, configure an equivalent finite automaton

    X→ bbY | aY | ϵ

    Y→ aYY | b | Z

    Z→ abZ | X | a

17. Write a grammar in Bakus Naur Form for the signed integers.

18. Verify that grammar defining language of palindromes over {a,b} is a CFG.

19. Define the closure properties of context free grammar.

❑❑❑