Chapter – 2

Regular Language:

Regular expression:

- The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. It is the most effective way to represent any language.
- The languages accepted by some regular expression are referred to as Regular languages.
- A regular expression can also be described as a sequence of pattern that defines a string.
- Regular expressions are used to match character combinations in strings. String searching algorithm used this pattern to find the operations on a string.

For instance:

In a regular expression, x* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx, ...}

In a regular expression, x^+ means one or more occurrence of x. It can generate {x, xx, xxx, xxxx, ...}

Operations on Regular Language

The various operations on regular language are:

Union: If L and M are two regular languages then their union L U M is also a union.

1. 1. L U M = {s | s is in L or s is in M}

Intersection: If L and M are two regular languages then their intersection is also an intersection.

1. 1. L ∩ M = {st | s is in L and t is in M}

Kleen closure: If L is a regular language then its Kleen closure L1* will also be a regular language.

1. 1. L* = Zero or more occurrence of language L.

Example 1:

Write the regular expression for the language accepting all combinations of a's, over the set ∑ = {a}

Solution:

All combinations of a's means a may be zero, single, double and so on. If a is appearing zero times, that means a null string. That is we expect the set of {ε, a, aa, aaa, ...}. So we give a regular expression for this as:

1. R = a*

That is Kleen closure of a.

Example 2:

Write the regular expression for the language accepting all combinations of a's except the null string, over the set ∑ = {a}

Solution:

The regular expression has to be built for the language

1. L = {a, aa, aaa, ...}

This set indicates that there is no null string. So we can denote regular expression as:

R = a^+

Example 3:

Write the regular expression for the language accepting all the string containing any number of a's and b's.

Solution:

The regular expression will be:

1. r.e. = (a + b)*

This will give the set as L = {ε, a, aa, b, bb, ab, ba, aba, bab, ...}, any combination of a and b.

The (a + b)* shows any combination with a and b even a null string.

Examples of Regular Expression

Example 1:

Write the regular expression for the language accepting all the string which are starting with 1 and ending with 0, over ∑ = {0, 1}.

Solution:

In a regular expression, the first symbol should be 1, and the last symbol should be 0. The r.e. is as follows:

1. R = 1 (0+1)* 0

Example 2:

Write the regular expression for the language starting and ending with a and having any having any combination of b's in between.

Solution:

The regular expression will be:

1. R = a b* b

Example 3:

Write the regular expression for the language starting with a but not having consecutive b's.

Solution: The regular expression has to be built for the language:

1. L = {a, aba, aab, aba, aaa, abab, ...}

The regular expression for the above language is:

1. R = {a + ab}*

Example 4:

Write the regular expression for the language accepting all the string in which any number of a's is followed by any number of b's is followed by any number of c's.

Solution: As we know, any number of a's means a* any number of b's means b*, any number of c's means c*. Since as given in problem statement, b's appear after a's and c's appear after b's. So the regular expression could be:

1. R = a* b* c*

Example 5:

Write the regular expression for the language over ∑ = {0} having even length of the string.

Solution:

The regular expression has to be built for the language:

1. L = {ε, 00, 0000, 000000, ...}

The regular expression for the above language is:

1. R = (00)*

Example 6:

Write the regular expression for the language having a string which should have atleast one 0 and alteast one 1.

Solution:

The regular expression will be:

1. R = [(0 + 1)* 0 (0 + 1)* 1 (0 + 1)*] + [(0 + 1)* 1 (0 + 1)* 0 (0 + 1)*]

Example 7:

Describe the language denoted by following regular expression

1. r.e. = (b* (aaa)* b*)*

Solution:

The language can be predicted from the regular expression by finding the meaning of it. We will first split the regular expression as:

r.e. = (any combination of b's) (aaa)* (any combination of b's)

L = {The language consists of the string in which a's appear triples, there is no restriction on the number of b's}

Example 8:

Write the regular expression for the language L over ∑ = {0, 1} such that all the string do not contain the substring 01.

Solution:

The Language is as follows:

1. L = {ε, 0, 1, 00, 11, 10, 100, ...}

The regular expression for the above language is as follows:

1. R = (1* 0*)

Example 9:

Write the regular expression for the language containing the string over {0, 1} in which there are at least two occurrences of 1's between any two occurrences of 1's between any two occurrences of 0's.

Solution: At least two 1's between two occurrences of 0's can be denoted by (0111*0)*.

Similarly, if there is no occurrence of 0's, then any number of 1's are also allowed. Hence the r.e. for required language is:

1. R = (1 + (0111*0))*

Example 10:

Write the regular expression for the language containing the string in which every 0 is immediately followed by 11.

Solution:

The regular expectation will be:

1. R = (011 + 1)*

Identity Rules:

Let P, Q and R be the regular expressions then the identity rules are as follows −

- εR=R ε=R
- ε*= ε ε is null string
- (Φ)*= ε Φ is empty string
- ΦR=R Φ= Φ
- Φ+R=R
- R+R=R
- RR*=R*R=R+
- (R*)*=R*
- E+RR*=R*
- (P+Q)R=PR+QR
- (P+Q)*=(P*Q*)*=(P*+Q*)*
- R*(ε+R)=(ε+R)R*=R*
- (R+ε)*=R*
- E+R*=R*
- (PQ)*P=P(QP)*
- R*R+R=R*R

Conversion of RE to FA

To convert the RE to FA, we are going to use a method called the subset method. This method is used to obtain FA from the given regular expression. This method is given below:

Step 1: Design a transition diagram for given regular expression, using NFA with ε moves.

Step 2: Convert this NFA with ε to NFA without ε.

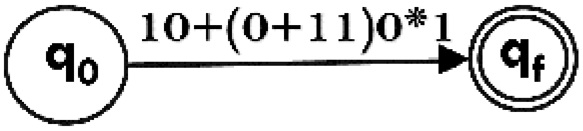
Step 3: Convert the obtained NFA to equivalent DFA.

Example 1:

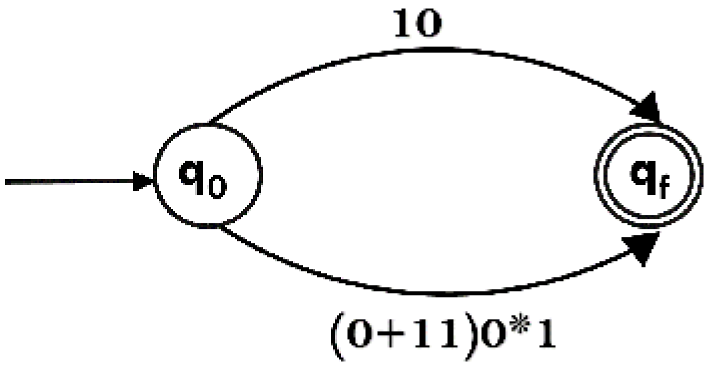
Design a FA from given regular expression 10 + (0 + 11)0* 1.

Solution: First we will construct the transition diagram for a given regular expression.

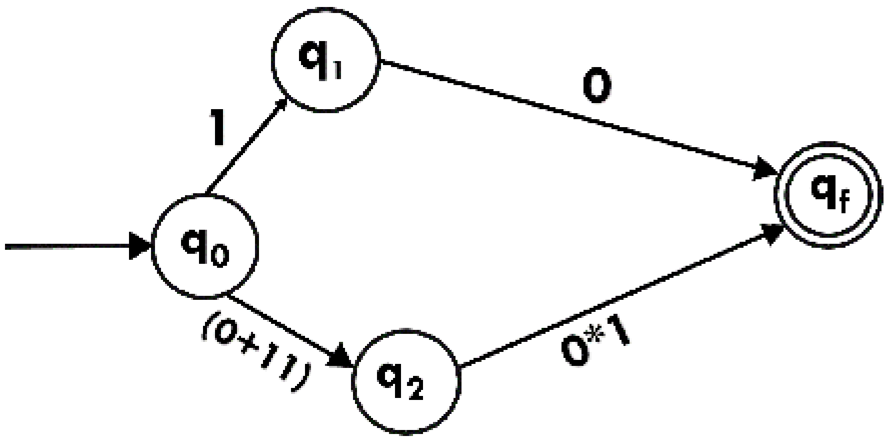
Step 1:



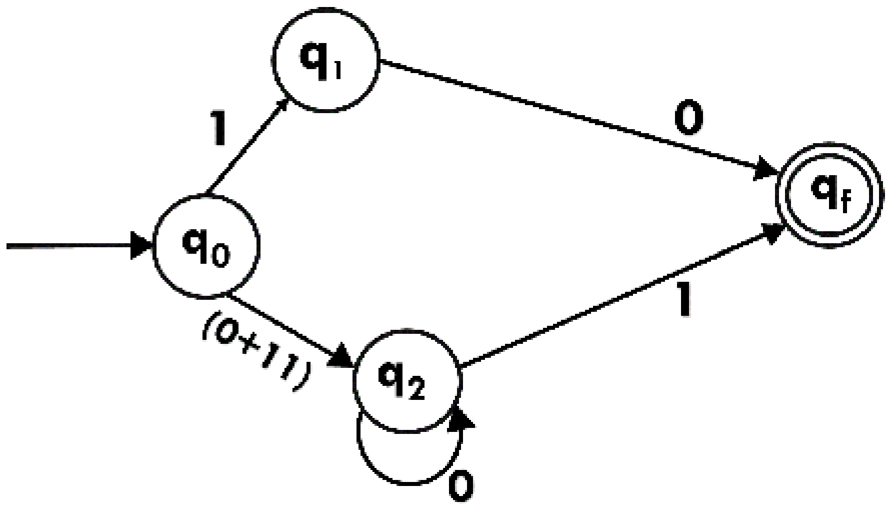
Step 2:



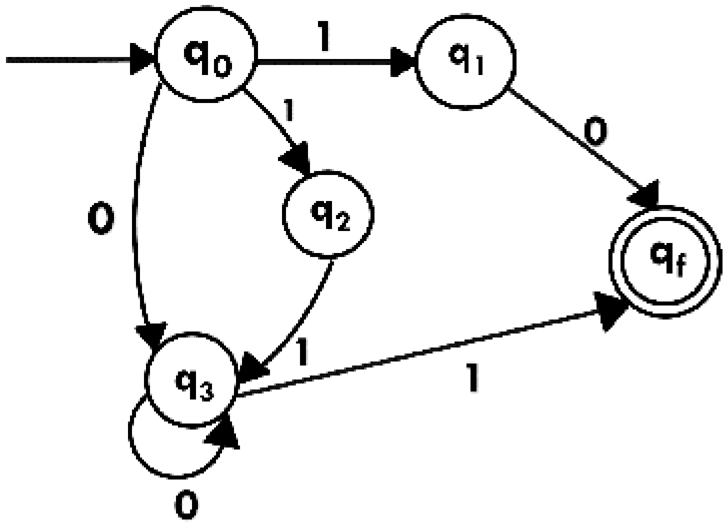
Step 3:



Step 4:



Step 5:



Now we have got NFA without ε. Now we will convert it into required DFA for that, we will first write a transition table for this NFA.

State	0	1
→q0	q3	{q1, q2}
q1	qf	ϕ

q2	ϕ	q3
q3	q3	qf
*qf	ϕ	ϕ

The equivalent DFA will be:

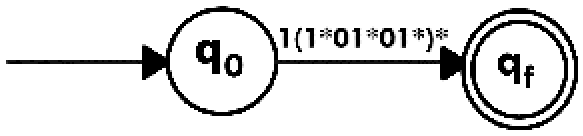
State	**0**	**1**
→[q0]	[q3]	[q1, q2]
[q1]	[qf]	ϕ
[q2]	ϕ	[q3]
[q3]	[q3]	[qf]
[q1, q2]	[qf]	[qf]
*[qf]	ϕ	ϕ

Example 2:

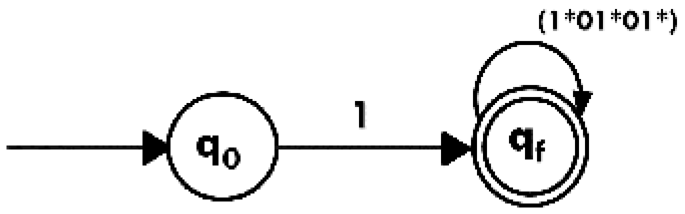
Design a NFA from given regular expression 1 (1* 01* 01*)*.

Solution: The NFA for the given regular expression is as follows:

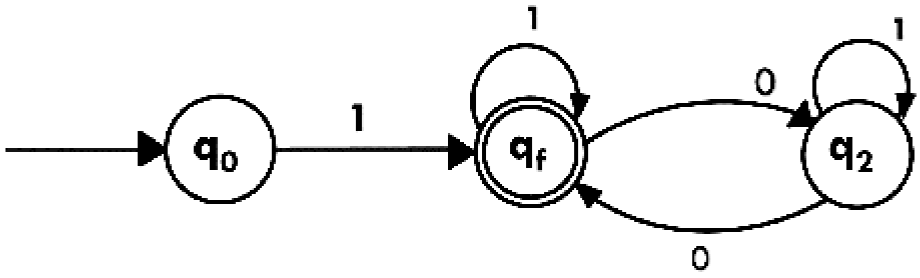
Step 1:


$$\begin{tikzcd} {} \arrow[r] & q_0 \arrow[r, "1(1^*01^*01^*)^*"] & \boxed{q_f} \end{tikzcd}$$

Step 2:



Step 3:



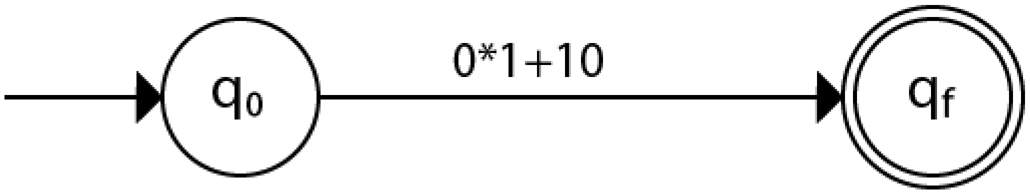
Example 3:

Construct the FA for regular expression 0*1 + 10.

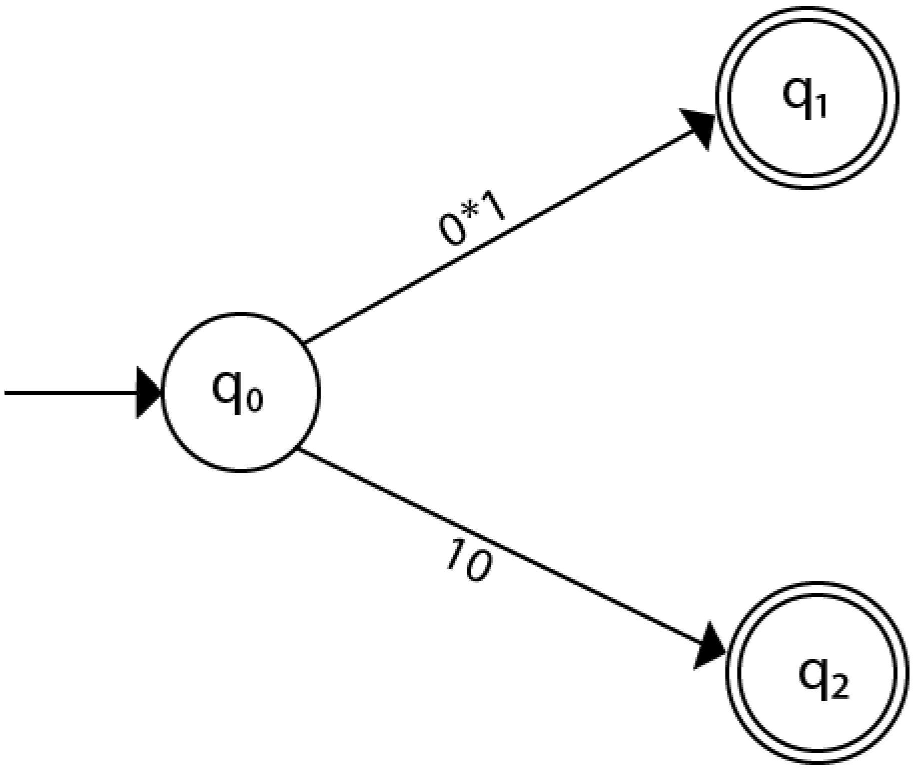
Solution:

We will first construct FA for R = 0*1 + 10 as follows:

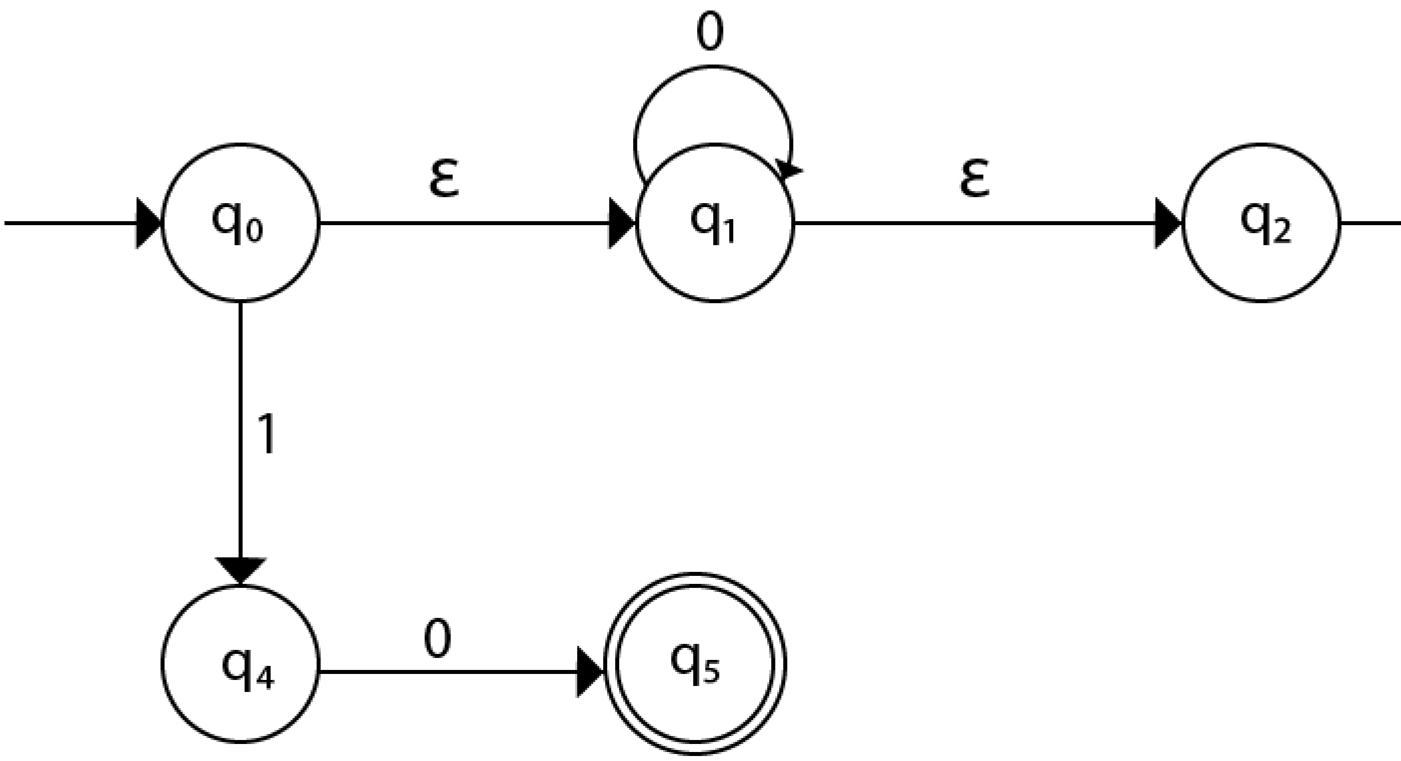
Step 1:



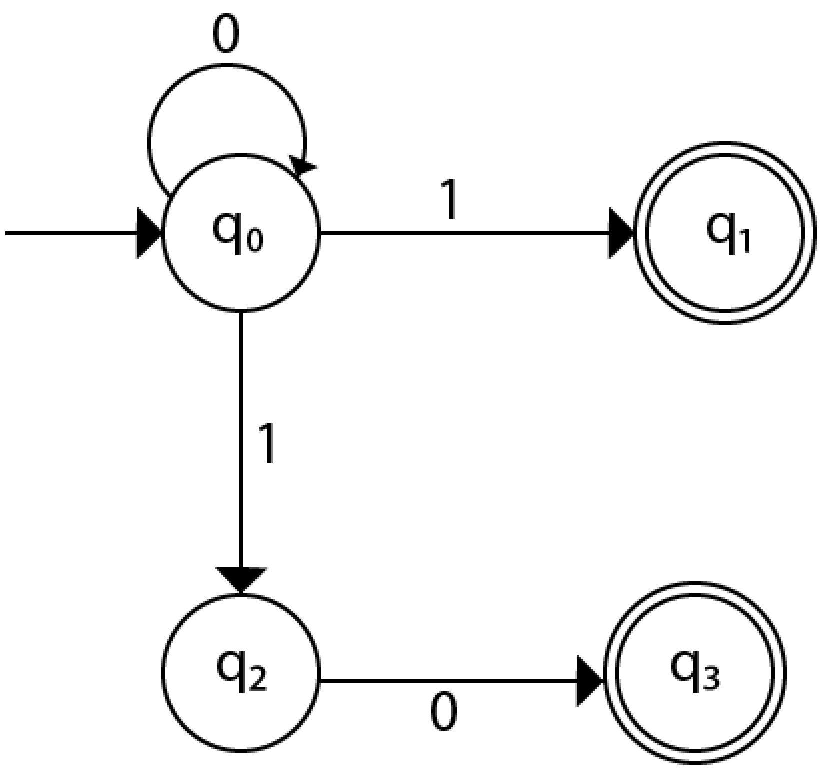
Step 2:



Step 3:



Step 4:



Conversion of Finite automata to regular expression:

Arden's Theorem

The Arden's Theorem is useful for checking the equivalence of two regular expressions as well as in the conversion of DFA to a regular expression.

Let us see its use in the conversion of DFA to a regular expression.

Following algorithm is used to build the regular expression form given DFA.

1. Let q_1 be the initial state.

2. There are q_2, q_3, q_4 ....q_n number of states. The final state may be some q_j where j<= n.

3. Let α_{ji} represents the transition from q_j to q_i.

4. Calculate q_i such that

$$q_i = \alpha_{ji} * q_j$$

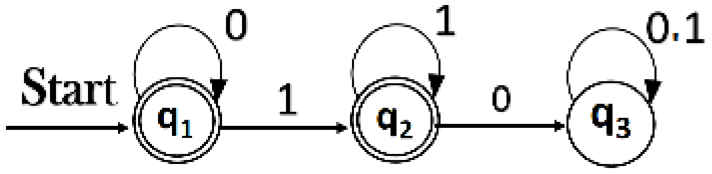
If q_j is a start state then we have:

$$q_i = \alpha_{ji} * q_j + \varepsilon$$

5. Similarly, compute the final state which ultimately gives the regular expression 'r'.

Example:

Construct the regular expression for the given DFA



Solution:

Let us write down the equations

q1 = q1 0 + ε

Since q1 is the start state, so ε will be added, and the input 0 is coming to q1 from q1 hence we write
State = source state of input × input coming to it

Similarly,

q2 = q1 1 + q2 1
q3 = q2 0 + q3 (0+1)

Since the final states are q1 and q2, we are interested in solving q1 and q2 only. Let us see q1 first

q1 = q1 0 + ε

We can re-write it as

q1 = ε + q1 0

Which is similar to R = Q + RP, and gets reduced to R = OP*.

Assuming R = q1, Q = ε, P = 0

We get

q1 = ε.(0)*
q1 = 0* (ε.R*= R*)

Substituting the value into q2, we will get

$q2 = 0^* 1 + q2\ 1$

$q2 = 0^* 1 (1)^*$ $(R = Q + RP \rightarrow Q P^*)$

The regular expression is given by

$r = q1 + q2$

$= 0^* + 0^* 1.1^*$

$r = 0^* + 0^* 1^+$ $(1.1^* = 1^+)$

Pumping lemma for Regular languages

- It gives a method for pumping (generating) many substrings from a given string.
- In other words, we say it provides means to break a given long input string into several substrings.
- Lt gives necessary condition(s) to prove a set of strings is not regular.

Theorem

For any regular language L, there exists an integer P, such that for all w in L

$|w|>=P$

We can break w into three strings, w=xyz such that.

(1)lxyl < P

(2)lyl > 1

(3)for all k>= 0: the string xy^kz is also in L

Application of pumping lemma

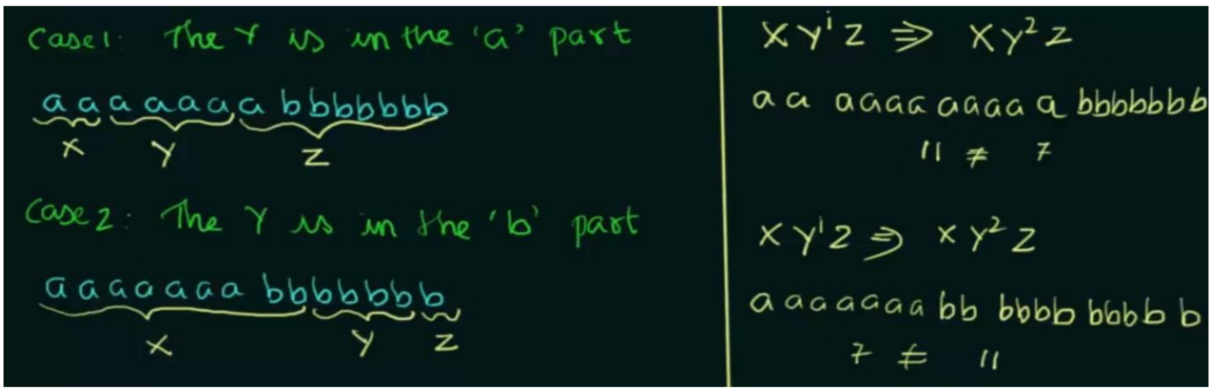
Pumping lemma is to be applied to show that certain languages are not regular.

It should never be used to show a language is regular.

- If L is regular, it satisfies the Pumping lemma.
- If L does not satisfy the Pumping Lemma, it is not regular.

Steps to prove that a language is not regular by using PLare as follows−

- step 1 − We have to assume that L is regular
- step 2 − So, the pumping lemma should hold for L.
- step 3 − It has to have a pumping length (say P).
- step 4 − All strings longer that P can be pumped |w|>=p.
- step 5 − Now find a string 'w' in L such that |w|>=P
- step 6 − Divide w into xyz.
- step 7 − Show that $xy^iz \notin L$ for some i.
- step 8 − Then consider all ways that w can be divided into xyz.
- step 9 − Show that none of these can satisfy all the 3 pumping conditions at same time.
- step 10 − w cannot be pumped = CONTRADICTION.



Closure Properties of regular set:

Closure properties on regular languages are defined as certain operations on regular language that are guaranteed to produce regular language. Closure refers to some operation on a language, resulting in a new language that is of the same “type” as originally operated on i.e., regular. Regular languages are closed under the following operations:

Consider that L and M are regular languages

1. **Kleen Closure:** RS is a regular expression whose language is L, M. R* is a regular expression whose language is L*.
2. **Positive closure:** RS is a regular expression whose language is L, M.   is a regular expression whose language is   .
3. **Complement:** The complement of a language L (with respect to an alphabet   such that   contains L) is   –L. Since   is surely regular, the complement of a regular language is always regular.

4. **Reverse Operator:** Given language L, is the set of strings whose reversal is in L. Example: L = {0, 01, 100}; ={0, 10, 001}. **Proof:** Let E be a regular expression for L. We show how to reverse E, to provide a regular expression for .
5. **Union:** Let L and M be the languages of regular expressions R and S, respectively.Then R+S is a regular expression whose language is(L U M).
6. **Intersection:** Let L and M be the languages of regular expressions R and S, respectively then it a regular expression whose language is L intersection M. **proof:** Let A and B be DFA's whose languages are L and M, respectively. Construct C, the product automaton of A and B make the final states of C be the pairs consisting of final states of both A and B.
7. **Set Difference operator:** If L and M are regular languages, then so is L – M = strings in L but not M. **Proof:** Let A and B be DFA's whose languages are L and M, respectively. Construct C, the product automaton of A and B make the final states of C be the pairs, where A-state is final but B-state is not.

Context free grammars and languages:

Context-Free Grammar (CFG)

CFG stands for context-free grammar. It is is a formal grammar which is used to generate all possible patterns of strings in a given formal language. Context-free grammar G can be defined by four tuples as:

1. G = (V, T, P, S)

Where,

G is the grammar, which consists of a set of the production rule. It is used to generate the string of a language.

T is the final set of a terminal symbol. It is denoted by lower case letters.

V is the final set of a non-terminal symbol. It is denoted by capital letters.

P is a set of production rules, which is used for replacing non-terminals symbols(on the left side of the production) in a string with other terminal or non-terminal symbols(on the right side of the production).

S is the start symbol which is used to derive the string. We can derive the string by repeatedly replacing a non-terminal by the right-hand side of the production until all non-terminal have been replaced by terminal symbols.

Example 1:

Construct the CFG for the language having any number of a's over the set ∑= {a}.

Solution:

As we know the regular expression for the above language is

1. r.e. = a*

Production rule for the Regular expression is as follows:

1. S → aS   rule 1
2. S → ε   rule 2

Now if we want to derive a string "aaaaaa", we can start with start symbols.

1. S
2. aS
3. aaS   rule 1
4. aaaS   rule 1
5. aaaaS   rule 1
6. aaaaaS   rule 1
7. aaaaaaS   rule 1
8. aaaaaaε   rule 2
9. aaaaaa

The r.e. = a* can generate a set of string {ε, a, aa, aaa,.....}. We can have a null string because S is a start symbol and rule 2 gives S → ε.

Example 2:

Construct a CFG for the regular expression (0+1)*

Solution:

The CFG can be given by,

1. Production rule (P):
2. S → 0S | 1S

3. S → ε

The rules are in the combination of 0's and 1's with the start symbol. Since (0+1)* indicates {ε, 0, 1, 01, 10, 00, 11, ...}. In this set, ε is a string, so in the rule, we can set the rule S → ε.

Example 3:

Construct a CFG for a language L = {wcwR | where w € (a, b)*}.

Solution:

The string that can be generated for a given language is {aacaa, bcb, abcba, bacab, abbcbba, ...}

The grammar could be:

1. S → aSa   rule 1
2. S → bSb   rule 2
3. S → c   rule 3

Now if we want to derive a string "abbcbba", we can start with start symbols.

1. S → aSa
2. S → abSba   from rule 2
3. S → abbSbba   from rule 2
4. S → abbcbba   from rule 3

Thus any of this kind of string can be derived from the given production rules.

Example 4:

Construct a CFG for the language L = $a^n b^{2n}$ where n>=1.

Solution:

The string that can be generated for a given language is {abb, aabbbb, aaabbbbbb....}.

The grammar could be:

1. S → aSbb | abb

Now if we want to derive a string "aabbbb", we can start with start symbols.

1. S → aSbb
2. S → aabbbb

Context-free languages (CFLs) are generated by context-free grammars. The set of all context-free languages is identical to the set of languages accepted by pushdown automata, and the set of regular languages is a subset of context-free languages. An inputed language is accepted by a computational model if it runs through the model and ends in an accepting final state. All regular languages are context-free languages, but not all context-free languages are regular. Most arithmetic expressions are generated by context-free grammars, and are therefore, context-free languages. Context-free languages and context-free grammars have applications in computer science and linguistics such as natural language processing and computer language design.

Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing, we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be placed with production rule.

1. Leftmost Derivation:

In the leftmost derivation, the input is scanned and replaced with the production rule from left to right. So in leftmost derivation, we read the input string from left to right.

Example:

Production rules:

1. E = E + E
2. E = E - E
3. E = a | b

Input

1. a - b + a

The leftmost derivation is:

1. E = E + E
2. E = E - E + E
3. E = a - E + E
4. E = a - b + E
5. E = a - b + a

2. Rightmost Derivation:

In rightmost derivation, the input is scanned and replaced with the production rule from right to left. So in rightmost derivation, we read the input string from right to left.

Example

Production rules:

1. E = E + E
2. E = E - E
3. E = a | b

Input

1. a - b + a

The rightmost derivation is:

1. E = E - E
2. E = E - E + E
3. E = E - E + a
4. E = E - b + a
5. E = a - b + a

When we use the leftmost derivation or rightmost derivation, we may get the same string. This type of derivation does not affect on getting of a string.

Examples of Derivation:

Example 1:

Derive the string "abb" for leftmost derivation and rightmost derivation using a CFG given by,

1. S → AB | ε
2. A → aB
3. B → Sb

Solution:

Leftmost derivation:

S

AB

$\boxed{aB}\ B$

$a\ \boxed{Sb}\ B$

$A\ \boxed{\varepsilon}\ bB$

$ab\ \boxed{Sb}$

$ab\ \boxed{\varepsilon}\ b$

abb

Rightmost derivation:

S

AB

$A\ \boxed{Sb}$

$A\ \boxed{\varepsilon}\ b$

$\boxed{aB}\ b$

$a\ \boxed{Sb}\ b$

$a\ \boxed{\varepsilon}\ bb$

abb

Example 2:

Derive the string "aabbabba" for leftmost derivation and rightmost derivation using a CFG given by,

1. S → aB | bA
2. S → a | aS | bAA
3. S → b | aS | aBB

Solution:

Leftmost derivation:

1. S
2. aB S → aB
3. aaBB B → aBB
4. aabB B → b
5. aabbS B → bS
6. aabbaB S → aB
7. aabbabS B → bS
8. aabbabbA S → bA
9. aabbabba A → a

Rightmost derivation:

1. S
2. aB S → aB
3. aaBB B → aBB
4. aaBbS B → bS
5. aaBbbA S → bA
6. aaBbba A → a
7. aabSbba B → bS
8. aabbAbba S → bA
9. aabbabba A → a

Example 3:

Derive the string "00101" for leftmost derivation and rightmost derivation using a CFG given by,

1. S → A1B
2. A → 0A | ε
3. B → 0B | 1B | ε

Solution:

Leftmost derivation:

1. S

2. A1B
3. 0A1B
4. 00A1B
5. 001B
6. 0010B
7. 00101B
8. 00101

Rightmost derivation:

1. S
2. A1B
3. A10B
4. A101B
5. A101
6. 0A101
7. 00A101
8. 00101

Derivation Tree

Derivation tree is a graphical representation for the derivation of the given production rules for a given CFG. It is the simple way to show how the derivation can be done to obtain some string from a given set of production rules. The derivation tree is also called a parse tree.

Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

A parse tree contains the following properties:

1. The root node is always a node indicating start symbols.
2. The derivation is read from left to right.
3. The leaf node is always terminal nodes.
4. The interior nodes are always the non-terminal nodes.

Example 1:

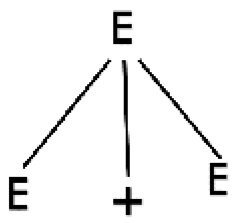
Production rules:

1. E = E + E
2. E = E * E
3. E = a | b | c

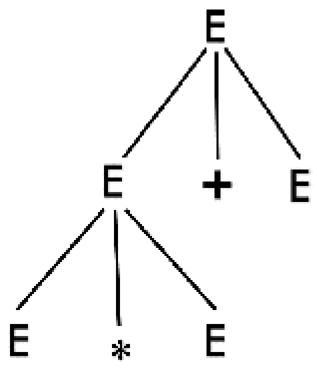
Input

1. a * b + c

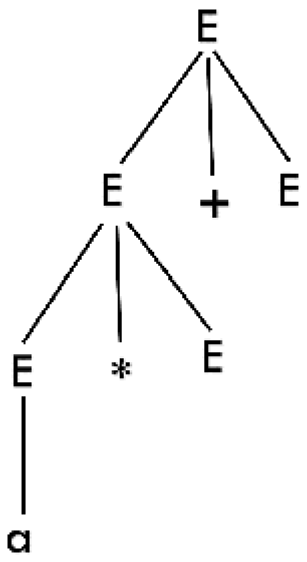
Step 1:



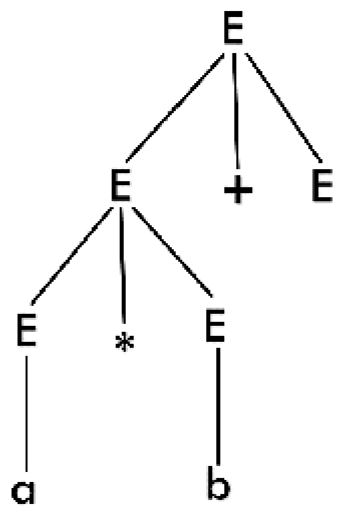
Step 2:



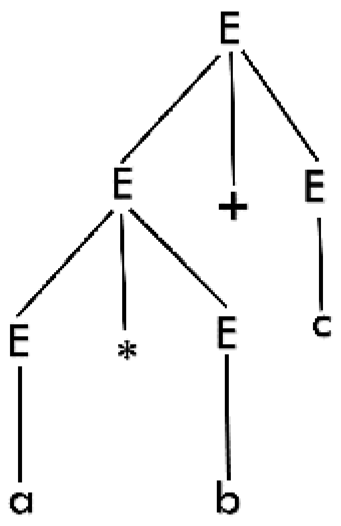
Step 2:



Step 4:



Step 5:



Note: We can draw a derivation tree step by step or directly in one step.

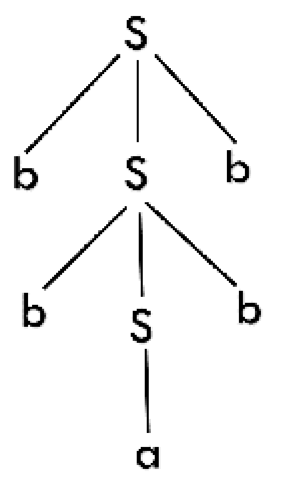
Example 2:

Draw a derivation tree for the string "bab" from the CFG given by

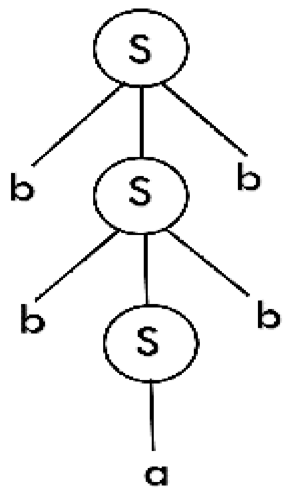
1. S → bSb | a | b

Solution:

Now, the derivation tree for the string "bbabb" is as follows:



The above tree is a derivation tree drawn for deriving a string bbabb. By simply reading the leaf nodes, we can obtain the desired string. The same tree can also be denoted by,



Left recursion and Left factoring:

Left recursion:

A Grammar G (V, T, P, S) is left recursive if it has a production in the form.

A → A α |β.

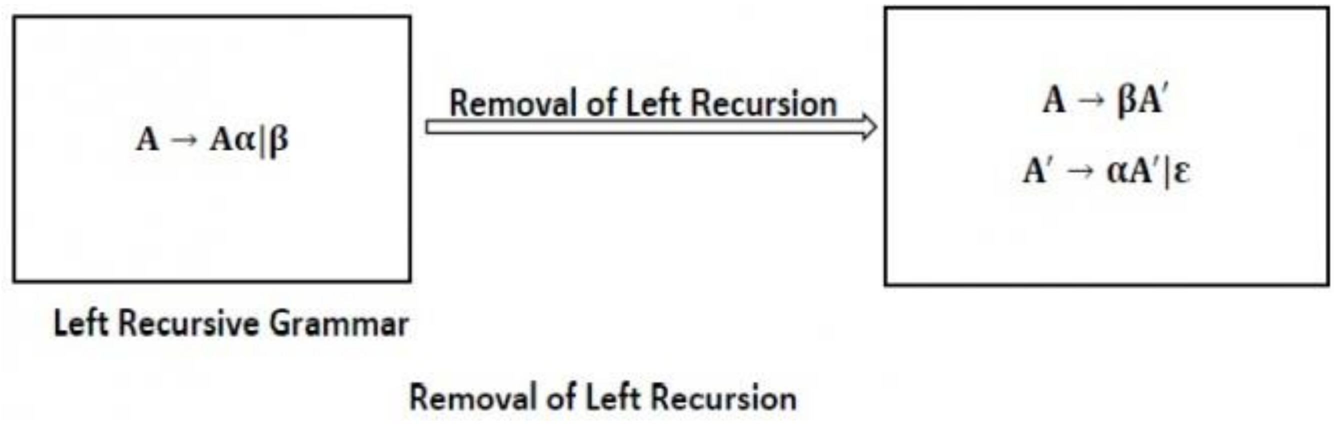
The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

A → βA′

$A \rightarrow \alpha A' | \epsilon$

Elimination of Left Recursion

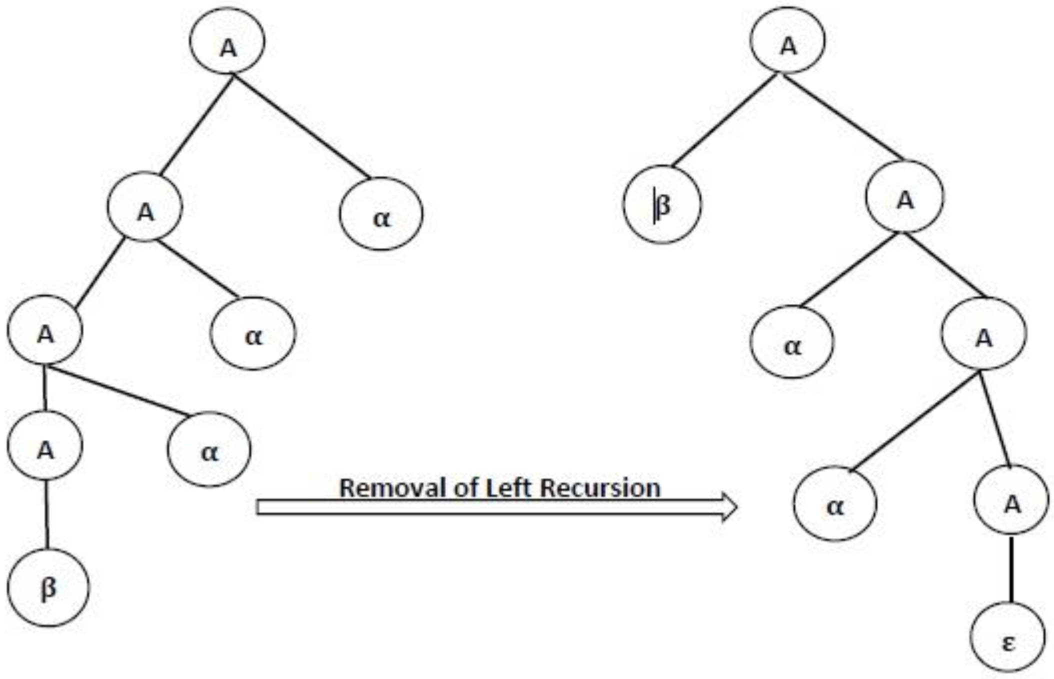
Left Recursion can be eliminated by introducing new non-terminal A such that.


$A \rightarrow A\alpha|\beta$
Removal of Left Recursion
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A'|\varepsilon$
Left Recursive Grammar

Removal of Left Recursion

This type of recursion is also called **Immediate Left Recursion**.

In Left Recursive Grammar, expansion of A will generate Aα, Aαα, Aααα at each step, causing it to enter into an infinite loop



The general form for left recursion is

$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots . | A\alpha_m | \beta_1 | \beta_2 | \ldots . . \beta_n$

can be replaced by

$A \rightarrow \beta_1 A' | \beta_2 A' | \; \ldots\ldots \; | \; \ldots\ldots \; | \beta_n A'$

$A \rightarrow \alpha_1 A' | \alpha_2 A' | \; \ldots\ldots \; | \alpha_m A' | \varepsilon$

Example1 − Consider the Left Recursion from the Grammar.

$E \rightarrow E + T | T$

$T \rightarrow T * F | F$

$F \rightarrow (E) | id$

Eliminate immediate left recursion from the Grammar.

Solution

Comparing $E \rightarrow E + T | T$ with $A \rightarrow A\,\alpha \; | \beta$

E	→	E	+T	|	T
A	→	A	α	|	B

∴ $A = E$, $\alpha = +T$, $\beta = T$

∴ $A \rightarrow A\,\alpha \; | \beta$ is changed to $A \rightarrow \beta A'$ and $A' \rightarrow \alpha\, A' | \varepsilon$

∴ $A \rightarrow \beta A'$ means $E \rightarrow TE'$

$A' \rightarrow \alpha\, A' | \varepsilon$ means $E' \rightarrow +TE' | \varepsilon$

Comparing $T \rightarrow T * F | F$ with $A \rightarrow A\alpha | \beta$

T	→	T	*F	|	F
A	→	A	α	|	β

∴ $A = T$, $\alpha = * F$, $\beta = F$

∴ $A \rightarrow \beta\, A'$ means $T \rightarrow FT'$

$A \rightarrow \alpha\, A' | \varepsilon$ means $T' \rightarrow * FT' | \varepsilon$

Production $F \rightarrow (E)|id$ does not have any left recursion

∴ Combining productions 1, 2, 3, 4, 5, we get

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

Example2 – Eliminate the left recursion for the following Grammar.

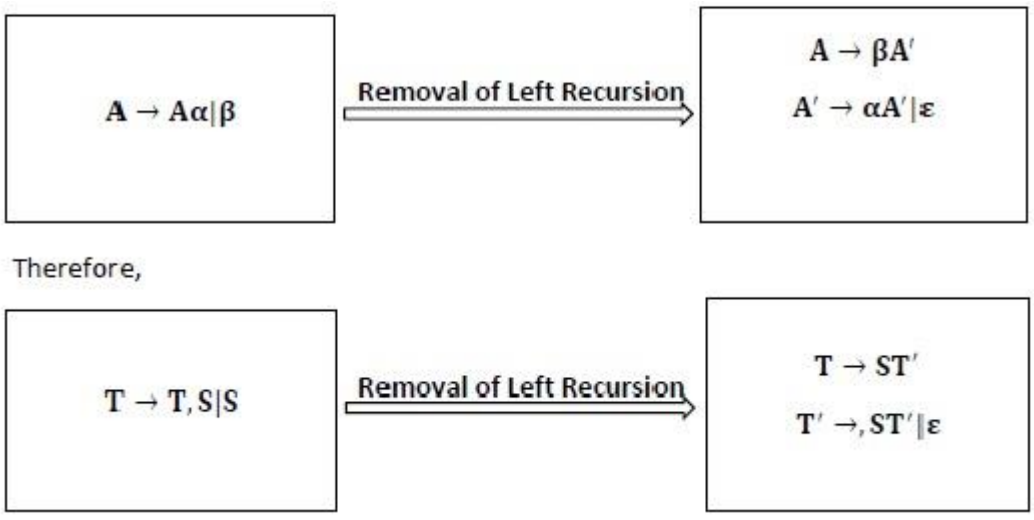
$S \rightarrow a | \wedge | (T)$

$T \rightarrow T, S | S$

Solution

We have immediate left recursion in T-productions.

Comparing $T \rightarrow T, S|S$ With $A \rightarrow A\ \alpha \mid \beta$ where $A = T$, $\alpha =, S$ and $\beta = S$



∴ Complete Grammar will be

$S \rightarrow a | \wedge(T)$
$T \rightarrow ST'$
$T' \rightarrow ,ST' \mid \varepsilon$

Example3 – Eliminate the left recursion from the grammar

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid id$

Solution

The production after removing the left recursion will be

$E \to TE'$

$E' \to +TE' \mid \in$

$T \to FT'$

$T' \to * FT' \mid \in$

$F \to (E) \mid id$

Example4 − Remove the left recursion from the grammar

$E \to E(T) \mid T$

$T \to T(F) \mid F$

$F \to id$

Solution

Eliminating immediate left-recursion among all Aα productions, we obtain

$E \to TE'$

$E \to (T)E' \mid \varepsilon$

$T \to FT'$

$T' \to (F)T' \mid \varepsilon$

$F \to id$

Left Factoring:
Left factored grammar in compiler design

Grammar in the following form will be considered to be having left factored-

S ⇒ aX | aY | aZ

S, X, Y, and **Z** are non-terminal symbols, and **a** is terminal. So left, factored grammar is having multiple productions with the same prefix or starting with the same symbol. In the example, ***S*** **⇒ aX,** ***S*** **⇒ aY,** and ***S*** **⇒ aZ** are three different productions with the same non–terminal symbol on the left-hand side, and the productions have a common prefix **a.** Hence the above grammar is left-factored.

Suppose the grammar is in the form:

A ⇒ αβ1 | αβ2 | αβ3 | …… | αβn | γ

Where **A** is a non-terminal and **α** is the common prefix.
We will separate those productions with a common prefix and then add a new production rule in which the new non-terminal we introduced will derive those productions with a common prefix.

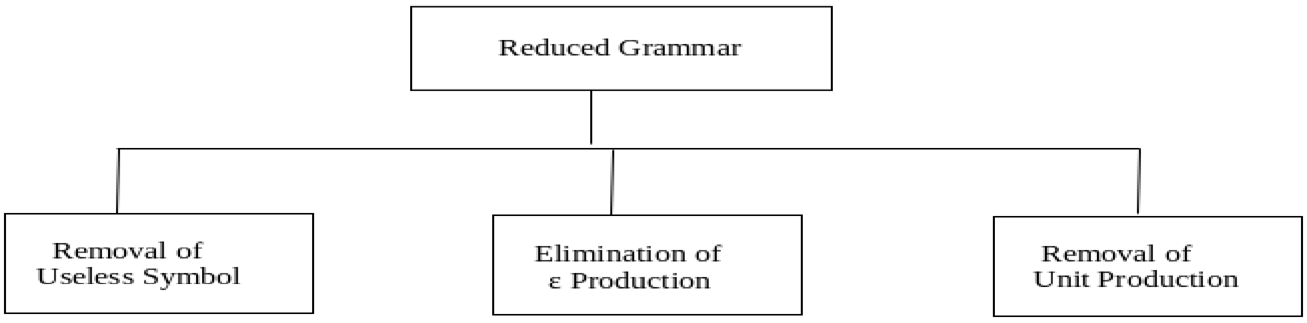
A ⇒ αA`
A` ⇒ β1 | β2 | β3 | …… | βn

The top-down parser can easily parse this grammar to derive a given string. So this is how left factoring in compiler design is performed on a given grammar.

Minimization of Context Free Grammar:

All the grammar are not always optimized that means the grammar may consist of some extra symbols(non-terminal). Having extra symbols, unnecessary increase the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols. The properties of reduced grammar are given below:

1. Each variable (i.e. non-terminal) and each terminal of G appears in the derivation of some word in L.
2. There should not be any production as X → Y where X and Y are non-terminal.
3. If ε is not in the language L then there need not to be the production X → ε.


Reduced Grammar
Removal of Useless Symbol
Elimination of ε Production
Removal of Unit Production

Removal of Useless Symbols

A symbol can be useless if it does not appear on the right-hand side of the production rule and does not take part in the derivation of any string. That symbol is known as a useless symbol. Similarly, a variable can be useless if it does not take part in the derivation of any string. That variable is known as a useless variable.

For Example:

1. T → aaB | abA | aaT
2. A → aA
3. B → ab | b
4. C → ad

In the above example, the variable 'C' will never occur in the derivation of any string, so the production C → ad is useless. So we will eliminate it, and the other productions are written in such a way that variable C can never reach from the starting variable 'T'.

Production A → aA is also useless because there is no way to terminate it. If it never terminates, then it can never produce a string. Hence this production can never take part in any derivation.

To remove this useless production A → aA, we will first find all the variables which will never lead to a terminal string such as variable 'A'. Then we will remove all the productions in which the variable 'B' occurs.

Elimination of ε Production

The productions of type S → ε are called ε productions. These type of productions can only be removed from those grammars that do not generate ε.

Step 1: First find out all nullable non-terminal variable which derives ε.

Step 2: For each production A → a, construct all production A → x, where x is obtained from a by removing one or more non-terminal from step 1.

Step 3: Now combine the result of step 2 with the original production and remove ε productions.

Example:

Remove the production from the following CFG by preserving the meaning of it.

1. S → XYX
2. X → 0X | ε
3. Y → 1Y | ε

Solution:

Now, while removing ε production, we are deleting the rule X → ε and Y → ε. To preserve the meaning of CFG we are actually placing ε at the right-hand side whenever X and Y have appeared.

Let us take

1. S → XYX

If the first X at right-hand side is ε. Then

1. S → YX

Similarly if the last X in R.H.S. = ε. Then

1. S → XY

If Y = ε then

1. S → XX

If Y and X are ε then,

1. S → X

If both X are replaced by ε

1. S → Y

Now,

1. S → XY | YX | XX | X | Y

Now let us consider

1. X → 0X

If we place ε at right-hand side for X then,

1. X → 0
2. X → 0X | 0

Similarly Y → 1Y | 1

Collectively we can rewrite the CFG with removed ε production as

1. S → XY | YX | XX | X | Y
2. X → 0X | 0
3. Y → 1Y | 1

Removing Unit Productions

The unit productions are the productions in which one non-terminal gives another non-terminal. Use the following steps to remove unit production:

Step 1: To remove X → Y, add production X → a to the grammar rule whenever Y → a occurs in the grammar.

Step 2: Now delete X → Y from the grammar.

Step 3: Repeat step 1 and step 2 until all unit productions are removed.

For example:

1. S → 0A | 1B | C
2. A → 0S | 00
3. B → 1 | A
4. C → 01

Solution:

S → C is a unit production. But while removing S → C we have to consider what C gives. So, we can add a rule to S.

1. S → 0A | 1B | 01

Similarly, B → A is also a unit production so we can modify it as

1. B → 1 | 0S | 00

Thus finally we can write CFG without unit production as

1. S → 0A | 1B | 01
2. A → 0S | 00
3. B → 1 | 0S | 00
4. C → 01

Chomsky's Normal Form (CNF)

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

- Start symbol generating ε. For example, A → ε.
- A non-terminal generating two non-terminals. For example, S → AB.
- A non-terminal generating a terminal. For example, S → a.

For example:

1. G1 = {S → AB, S → c, A → a, B → b}
2. G2 = {S → aA, A → a, B → c}

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF. However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as S → aZ contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

Steps for converting CFG into CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

1. S1 → S

Where S1 is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:

1. S → RA
2. R → a

Step 4: Eliminate RHS with more than two non-terminals. For example, S → ASB can be decomposed as:

1. S → RS
2. R → AS

Example:

Convert the given CFG to CNF. Consider the given grammar G1:

1. S → a | aA | B
2. A → aBB | ε
3. B → Aa | b

Solution:

Step 1: We will create a new production S1 → S, as the start symbol S appears on the RHS. The grammar will be:

1. S1 → S
2. S → a | aA | B
3. A → aBB | ε
4. B → Aa | b

Step 2: As grammar G1 contains A → ε null production, its removal from the grammar yields:

1. S1 → S
2. S → a | aA | B
3. A → aBB
4. B → Aa | b | a

Now, as grammar G1 contains Unit production S → B, its removal yield:

1. S1 → S
2. S → a | aA | Aa | b
3. A → aBB

4. B → Aa | b | a

Also remove the unit production S1 → S, its removal from the grammar yields:

1. S0 → a | aA | Aa | b
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Step 3: In the production rule S0 → aA | Aa, S → aA | Aa, A → aBB and B → Aa, terminal a exists on RHS with non-terminals. So we will replace terminal a with X:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → XBB
4. B → AX | b | a
5. X → a

Step 4: In the production rule A → XBB, RHS has more than two symbols, removing it from grammar yield:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → RB
4. B → AX | b | a
5. X → a
6. R → XB

Hence, for the given grammar, this is the required CNF.

Greibach Normal Form (GNF)

GNF stands for Greibach normal form. A CFG(context free grammar) is in GNF(Greibach normal form) if all the production rules satisfy one of the following conditions:

- A start symbol generating ε. For example, S → ε.
- A non-terminal generating a terminal. For example, A → a.
- A non-terminal generating a terminal which is followed by any number of non-terminals. For example, S → aASB.

For example:

1. G1 = {S → aAB | aB, A → aA| a, B → bB | b}
2. G2 = {S → aAB | aB, A → aA | ε, B → bB | ε}

The production rules of Grammar G1 satisfy the rules specified for GNF, so the grammar G1 is in GNF. However, the production rule of Grammar G2 does not satisfy the rules specified for GNF as A → ε and B → ε contains ε(only start symbol can generate ε). So the grammar G2 is not in GNF.

Steps for converting CFG into GNF

Step 1: Convert the grammar into CNF.

If the given grammar is not in CNF, convert it into CNF. You can refer the following topic to convert the CFG into CNF: Chomsky normal form

Step 2: If the grammar exists left recursion, eliminate it.

If the context free grammar contains left recursion, eliminate it. You can refer the following topic to eliminate left recursion: Left Recursion

Step 3: In the grammar, convert the given production rule into GNF form.

If any production rule in the grammar is not in GNF form, convert it.

Example:

1. S → XB | AA
2. A → a | SA
3. B → b
4. X → a

Solution:

As the given grammar G is already in CNF and there is no left recursion, so we can skip step 1 and step 2 and directly go to step 3.

The production rule A → SA is not in GNF, so we substitute S → XB | AA in the production rule A → SA as:

1. S → XB | AA
2. A → a | XBA | AAA
3. B → b
4. X → a

The production rule S → XB and B → XBA is not in GNF, so we substitute X → a in the production rule S → XB and B → XBA as:

1. S → aB | AA
2. A → a | aBA | AAA
3. B → b
4. X → a

Now we will remove left recursion (A → AAA), we get:

1. S → aB | AA
2. A → aC | aBAC
3. C → AAC | ε
4. B → b
5. X → a

Now we will remove null production C → ε, we get:

1. S → aB | AA
2. A → aC | aBAC | a | aBA
3. C → AAC | AA
4. B → b
5. X → a

The production rule S → AA is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule S → AA as:

1. S → aB | aCA | aBACA | aA | aBAA
2. A → aC | aBAC | a | aBA
3. C → AAC
4. C → aCA | aBACA | aA | aBAA
5. B → b
6. X → a

The production rule C → AAC is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule C → AAC as:

S → aB | aCA | aBACA | aA | aBAA

1. A → aC | aBAC | a | aBA
2. C → aCAC | aBACAC | aAC | aBAAC
3. C → aCA | aBACA | aA | aBAA
4. B → b
5. X → a

Hence, this is the GNF form for the grammar G.

Pumping Lemma for CFG:

Lemma

If **L** is a context-free language, there is a pumping length **p** such that any string **w** $\in$ **L** of length $\geq$ **p** can be written as **w = uvxyz**, where **vy** $\neq \varepsilon$, **|vxy|** $\leq$ **p**, and for all **i** $\geq$ **0,** $uv^ixy^iz \in L$.

Applications of Pumping Lemma

Pumping lemma is used to check whether a grammar is context free or not. Let us take an example and show how it is checked.

Problem

Find out whether the language $L = \{x^ny^nz^n \mid n \geq 1\}$ is context free or not.

Solution

Let **L** is context free. Then, **L** must satisfy pumping lemma.

At first, choose a number **n** of the pumping lemma. Then, take z as $0^n1^n2^n$.

Break **z** into **uvwxy,** where

|vwx| $\leq$ **n and vx** $\neq \varepsilon$**.**

Hence **vwx** cannot involve both 0s and 2s, since the last 0 and the first 2 are at least (n+1) positions apart. There are two cases −

Case 1 − **vwx** has no 2s. Then **vx** has only 0s and 1s. Then **uwy**, which would have to be in **L**, has **n** 2s, but fewer than **n** 0s or 1s.

Case 2 − **vwx** has no 0s.

Here contradiction occurs.

Hence, **L** is not a context-free language.

CFL Closure Property

Context-free languages are **closed** under −

- Union
- Concatenation
- Kleene Star operation

Union

Let L_1 and L_2 be two context free languages. Then $L_1 \cup L_2$ is also context free.

Example

Let $L_1 = \{ a^n b^n , n > 0\}$. Corresponding grammar G_1 will have P: $S1 \rightarrow aAb|ab$

Let $L_2 = \{ c^m d^m , m \geq 0\}$. Corresponding grammar G_2 will have P: $S2 \rightarrow cBb|\ \varepsilon$

Union of L_1 and L_2, $L = L_1 \cup L_2 = \{ a^n b^n \} \cup \{ c^m d^m \}$

The corresponding grammar G will have the additional production $S \rightarrow S1 \mid S2$

Concatenation

If L_1 and L_2 are context free languages, then L_1L_2 is also context free.

Example

Union of the languages L_1 and L_2, $L = L_1L_2 = \{ a^n b^n c^m d^m \}$

The corresponding grammar G will have the additional production $S \rightarrow S1\ S2$

Kleene Star

If L is a context free language, then L* is also context free.

Example

Let $L = \{ a^n b^n , n \geq 0\}$. Corresponding grammar G will have P: $S \rightarrow aAb|\ \varepsilon$

Kleene Star $L_1 = \{ a^n b^n \}^*$

The corresponding grammar G_1 will have additional productions $S1 \rightarrow SS_1 \mid \varepsilon$

Context-free languages are **not closed** under −

- **Intersection** − If L1 and L2 are context free languages, then L1 ∩ L2 is not necessarily context free.
- **Intersection with Regular Language** − If L1 is a regular language and L2 is a context free language, then L1 ∩ L2 is a context free language.
- **Complement** − If L1 is a context free language, then L1' may not be context free.