SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF SCIENCE AND HUMANITIES

DEPARTMENT OF MATHEMATICS

UNIT – I – Formal Language and Automata – SMT1404

Introduction

Formal Languages

Automata Theory is an exciting, theoretical branch of computer science. It established its roots during the 20th Century, as mathematicians began developing - both theoretically and literally - machines which imitated certain features of man, completing calculations more quickly and reliably. The word **automaton** itself, closely related to the word "automation", denotes automatic processes carrying out the production of specific processes. Simply stated, automata theory deals with the logic of computation with respect to simple machines, referred to as **automata**. Through automata, computer scientists are able to understand how machines compute functions and solve problems and more importantly, what it means for a function to be defined as computable or for a question to be described as decidable.

Automatons are abstract models of machines that perform computations on an input by moving through a series of states or configurations. At each state of the computation, a transition function determines the next configuration on the basis of a finite portion of the present configuration. As a result, once the computation reaches an accepting configuration, it accepts that input.

The **major objective** of automata theory is to develop methods by which computer scientists can describe and analyze the dynamic behavior of discrete systems, in which signals are sampled periodically. The behavior of these discrete systems is determined by the way that the system is constructed from storage and combinational elements. Characteristics of such machines include:

Alphabets

An alphabet is a finite set of non-empty symbols. It is denoted by Σ.

1. Σ = {0,1}, the set of binary numbers.
2. Σ = {a, b, c, d,…,}, the set of all alphabets.

Strings

A string or word is a finite sequence of symbols chosen from some alphabet.

For example, the string w =10101 is a string over Σ = {0, 1}.

If ∑ = {a, b}, various strings that can be generated from Σ are {ab, aa, aaa, bb, bbb, ba, aba.....}.

Empty String

Empty string is a string with zero occurrences of symbols. It is denoted by ε.

Length of a String
The number of symbols in a string w is called the length of a string. It is denoted by |w|.

Examples −

If w = ‘cabcad’, |w|= 6

If |w|= 0, it is called an **empty string** (Denoted by **λ** or **ε**)

Powers of an Alphabet

If Σ is an alphabet, we can express the set of all strings of a certain length from that alphabet by using an exponential notation. We define Σ^k to be the set of strings of length k, each of whose symbols is in Σ.

Notations
The set of all strings over Σ is conveniently denoted Σ*

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$.
- $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

Example:

$\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$.

Languages

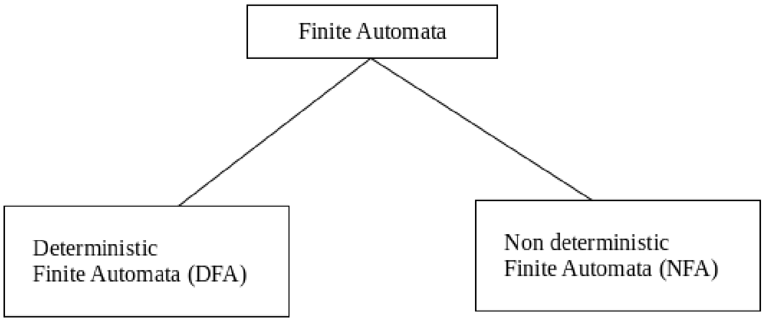
A set of strings all of which are chosen from some Σ^*, where Σ is a particular alphabet, is called a *language*. If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a *language over* Σ. Notice that a language over Σ need not include strings with all the symbols of Σ, so once we have established that L is a language over Σ, we also know it is a language over any alphabet that is a superset of Σ.

Finite Automata

1. Finite automata are used to recognize patterns.
2. It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
3. At the time of transition, the automata can either move to the next state or stay in the same state.
4. Finite automata have two states, **Accept state** or **Reject state**. When the input string is processed successfully, and the automata reached its final state, then it will accept.

Types of Automata:
There are two types of finite automata:

1. DFA(deterministic finite automata)
2. NFA(non-deterministic finite automata)



Note:

1. Every DFA is NFA, but NFA is not DFA.
2. There can be multiple final states in both NFA and DFA.
3. DFA is used in Lexical Analysis in Compiler.
4. NFA is more of a theoretical concept.

Deterministic Finite Automaton (DFA)

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called **Deterministic Automaton**. As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton.**

Formal Definition of a DFA

A DFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where −

- **Q** is a finite set of states.
- **Σ** is a finite set of symbols called the alphabet.
- **δ** is the transition function where δ: Q × ∑ → Q
- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

Transition Diagram

A *transition diagram* for a DFA $A = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

a) For each state in Q there is a node.

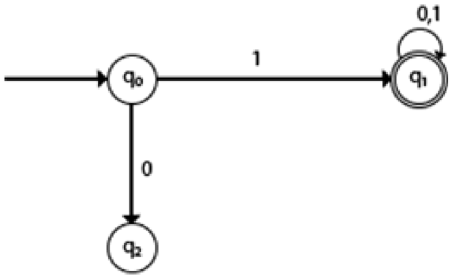
b) For each state q in Q and each input symbol a in Σ, let $\delta(q, a) = p$. Then the transition diagram has an arc from node q to node p, labeled a. If there are several input symbols that cause transitions from q to p, then the transition diagram can have one arc, labeled by the list of these symbols.

c) There is an arrow into the start state q_0, labeled *Start*. This arrow does not originate at any node.

d) Nodes corresponding to accepting states (those in F) are marked by a double circle. States not in F have a single circle.

Example 1:
DFA with ∑ = {0, 1} accepts all strings starting with 1.

Solution:



The finite automata can be represented using a transition graph. In the above diagram, the machine initially is in start state q_0 then on receiving input 1 the machine changes its state to q_1. From q0 on receiving 0, the machine changes its state to q_2, which is the dead state. From q_1 on receiving input 0, 1 the machine changes its state to q_1, which is the final state. The possible input strings that can be generated are 10, 11, 110, 101, 111.......
That means all string starts with 1.

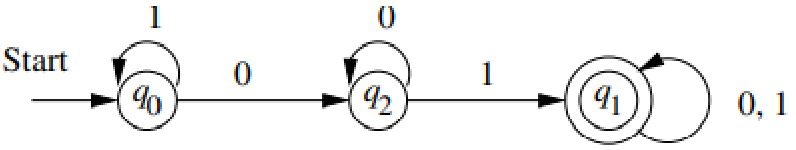
Transition Table
The transition table is basically a tabular representation of the transition function. It takes two arguments (a state and a symbol) and returns a state (the "next state").

A transition table is represented by the following things:

1. Columns correspond to input symbols.
2. Rows correspond to states.
3. Entries correspond to the next state.
4. The start state is denoted by an arrow with no source.
5. The accept state is denoted by a star.

Example 2

Find the transition table for the given automata



The transition table for the above diagram is given below

- The final state is indicated by double circles.

Acceptability by DFA

A string is accepted by a DFA/ iff the DFA/NDFA starting at the initial state ends in an accepting state (any of the final states) after reading the string wholly.

A string S is accepted by a DFA (Q, ∑, δ, q_0, F), iff **$\delta^*(q_0, S) \in F$**

The language **L** accepted by DFA is **$\{S \mid S \in \Sigma^* \text{ and } \delta^*(q_0, S) \in F\}$**

A string S′ is not accepted by a DFA (Q, ∑, δ, q_0, F), iff **$\delta^*(q_0, S') \notin F$**

The language L′ not accepted by DFA (Complement of accepted language L) is

$\{S \mid S \in \Sigma^* \text{ and } \delta^*(q_0, S) \notin F\}$

Problems

Problem: 1

Find the directed graph to the deterministic finite automaton M = (Q, Σ, q_0, δ, F) where

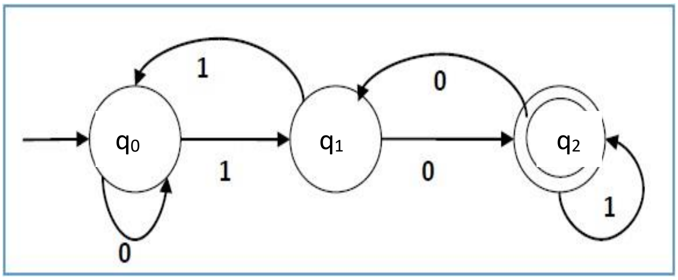
- Q = { q_0, q_1, q_2},
- Σ = {0, 1},
- q_0 = { q_0},
- F = { q_2}, and

Transition function δ as shown by the following table −

Inputs / States	0	1
→ q_0	q_0	q_1
q_1	q_2	q_0
q_2	q_1	q_2

Solution:

Its graphical representation would be as follows −

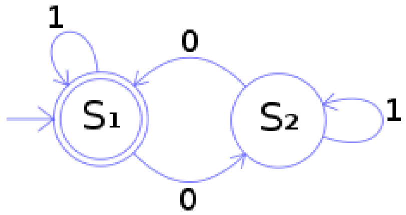


Problem: 2

Construct the language consists of an even number of 0s from the DFA M=(Q, Σ, δ, q_0,F) where Q={S_1,S_2} , Σ ={0,1}, q_0=S_1,F={S_1} and δ is defined by the following state transition table:

Inputs / States	0	1
→ S_1	S_2	S_1
S_2	S_1	S_2

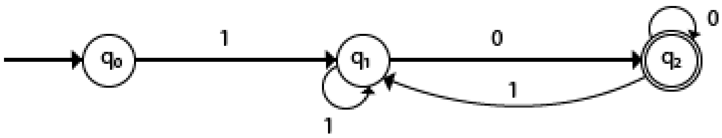
Solution:



The state S_1 represents that there has been an even number of 0s in the input so far, while S_2 signifies an odd number. A 1 in the input does not change the state of the automaton. When the input ends, the state will show whether the input contained an even number of 0s or not. If the input did contain an even number of 0s, M will finish in state S_1, an accepting state, so the input string will be accepted.

Problem: 3
Design a DFA with Σ= {0, 1} accepts those string which starts with 1 and ends with 0.

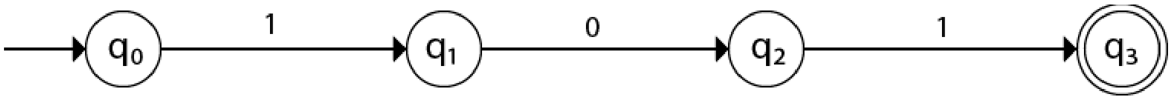
Solution:
Consider the DFA M = (Q, Σ, δ, q_0,) where Q={q_0, q_1, q_2} , Σ ={0,1}, q_0=S,F={q_2} with the transition diagram



The FA will have a start state q_0 from which only the edge with input 1 will go to the next state. In state q_1, if we read 1, we will be in state q_1, but if we read 0 at state q_1, we will reach to state q_2 which is the final state. In state q_2, if we read either 0 or 1, we will go to q_2 state or q_1 state respectively. Note that if the input ends with 0, it will be in the final state.

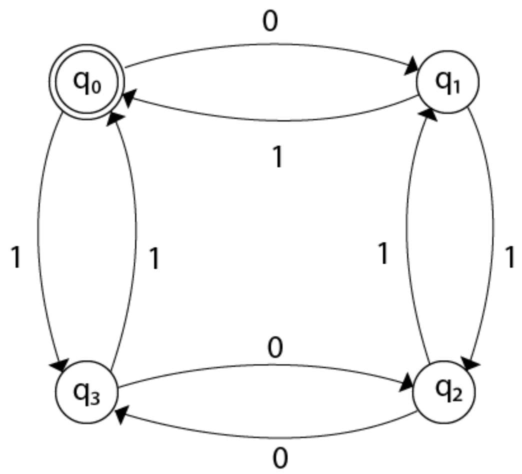
Problem 4:
Design a FA with ∑ = {0, 1} accepts the only input 101.

Solution:



Problem 5:
Design FA with ∑ = {0, 1} accepts even number of 0's and even number of 1's.

Solution: This FA will consider four different stages for input 0 and input 1. The stages could be:



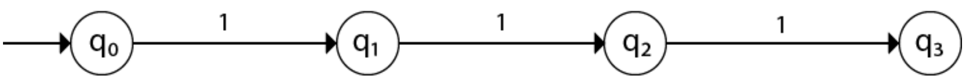
Here q_0 is a start state and the final state also. Note carefully that symmetry of 0's and 1's is maintained. We can associate meanings to each state as:

q_0: state of even number of 0's and even number of 1's.
q_1: state of odd number of 0's and even number of 1's.
q_2: state of odd number of 0's and odd number of 1's.
q_3: state of even number of 0's and odd number of 1's.

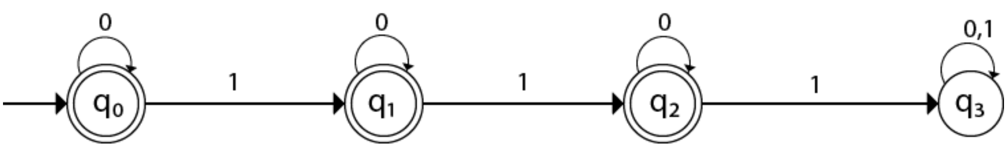
Problem 6:
Design a DFA L(M) = {w | w ε {0, 1}*} and W is a string that does not contain consecutive 1's.

Solution:

When three consecutive 1's occur the DFA will be:



Here two consecutive 1's or single 1 is acceptable, hence

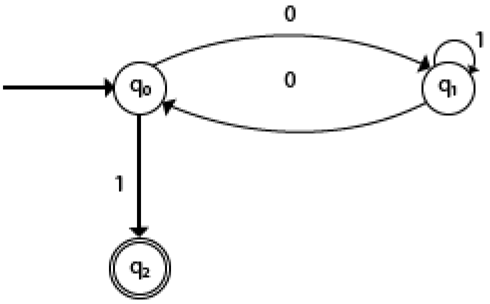


The stages q0, q1, q2 are the final states. The DFA will generate the strings that do not contain consecutive 1's like 10, 110, 101,..... etc.

Problem 7:
Design a FA with ∑ = {0, 1} accepts the strings with an even number of 0's followed by single 1.

Solution:

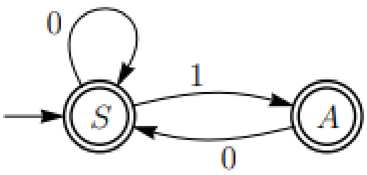
The DFA can be shown by a transition diagram as:



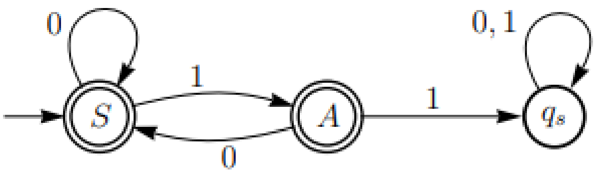
Problem 8:

Let us consider the finite automaton h{S, A}, {0, 1}, S, {S, A}, δi, where δ is the following partial transition function: δ(S, 0) = S δ(S, 1) = A δ(A, 0) = S.

This automaton is depicted below.



In order to get the equivalent finite automaton with a total transition function we consider the additional state q_s, called the **sink state**, and we stipulate that δ(A, 1) = qs ; δ(qs, 0) = qs; δ(qs, 1) = qs.

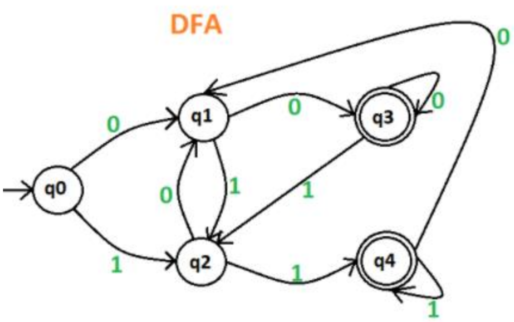


Problem for practice

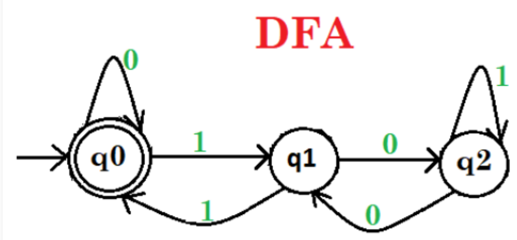
Problem 1:

Draw a deterministic and non-deterministic finite automate which accept 00 and 11 at the end of a string containing 0, 1 in it, e.g., 01010100 but not 000111010.

Solution:



Problem2:
Draw a deterministic finite automata which recognize a string containing binary representation 0, 1 in the form of multiple 3, e.g., 1001 but not 1000.
Solution:



Problem 3:

Construct a DFA for the set of string over {a, b} such that length of the string |w| is divisible by 3 i.e, |w| mod 3 = 0.

Problem 4:
Construction of a DFA for the set of string over {a, b} such that length of the string |w| is divisible by 2 i.e, |w| mod 2 = 0.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF SCIENCE AND HUMANITIES

DEPARTMENT OF MATHEMATICS

UNIT – II - Formal Language and Automata – SMT1404

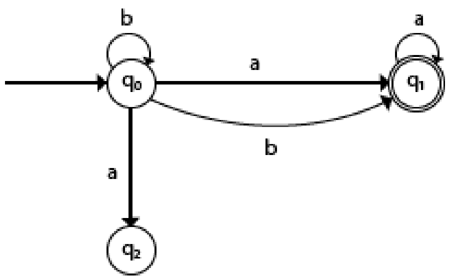
Introduction

Non-Deterministic Finite State Automata

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called **Non-deterministic Automaton**. As it has finite number of states, the machine is called **Non-deterministic Finite Machine** or **Non-deterministic Finite Automaton**.

1. NFA stands for non-deterministic finite automata. It is easy to construct an NFA than DFA for a given regular language.
2. The finite automata are called NFA when there exist many paths for specific input from the current state to the next state.
3. Every NFA is not DFA, but each NFA can be translated into DFA.
4. NFA is defined in the same way as DFA but with the following two exceptions, it contains multiple next states, and it contains ε transition.

In the following image, we can see that from state q_0 for input a, there are two next states q_1 and q_2, similarly, from q_0 for input b, the next states are q_0 and q_1. Thus it is not fixed or determined that with a particular input where to go next. Hence this FA is called non-deterministic finite automata.



Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where −

- **Q** is a finite set of states.
- ∑ is a finite set of symbols called the alphabets.
- **δ** is the transition function where $\delta: Q \times \sum \rightarrow 2^Q$

  (Here the power set of Q (2^Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)
- $\mathbf{q_0}$ is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

Graphical Representation of an NDFA: (same as DFA)

An NDFA is represented by digraphs called state diagram.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.

DFA	NDFA
The transition from a state is to a single particular next state for each input symbol. Hence it is called *deterministic*.	The transition from a state can be to multiple next states for each input symbol. Hence it is called *non-deterministic*.
Empty string transitions are not seen in DFA.	NDFA permits empty string transitions.
Backtracking is allowed in DFA	In NDFA, backtracking is not always possible.
Requires more space.	Requires less space.
A string is accepted by a DFA, if it transits to a final state.	A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state.

Acceptability by NDFA

A string is accepted by a NDFA iff the NDFA starting at the initial state ends in an accepting state (any of the final states) after reading the string wholly.

A string S is accepted by a NDFA (Q, ∑, δ, q_0, F), iff **δ*(q_0, S) ∈ F**

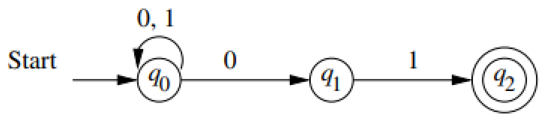
The language **L** accepted by NDFA is **{S | S ∈ ∑* and δ*(q_0, S) ∈ F}**

A string S′ is not accepted by a NDFA (Q, ∑, δ, q_0, F), iff **δ*(q_0, S′) ∉ F**

The language L′ not accepted by NDFA (Complement of accepted language L) is

{S | S ∈ ∑* and δ*(q_0, S) ∉ F}.

Problem1:
Consider the NDFA M = ({q_0,q_1, q_2}, {0,1}, δ, q_0, {q_0}) with the transition diagram



The transition table of the above NDFA is given below

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

Languages of NDFA

As we have suggested, an NFA accepts a string w if it is possible to make any sequence of choices of next state, while reading the characters of w, and go from the start state to any accepting state. The fact that other choices using the input symbols of w lead to a nonaccepting state, or do not lead to any state at all (i.e., the sequence of states "dies"), does not prevent w from being accepted by the NFA as a whole. Formally, if $A = (Q, \Sigma, \delta, q_0, F)$ is an NFA, then

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

That is, $L(A)$ is the set of strings w in Σ^* such that $\hat{\delta}(q_0, w)$ contains at least one accepting state.

Formal Definition of ε-NFA

We may represent an ϵ-NFA exactly as we do an NFA, with one exception: the transition function must include information about transitions on ϵ. Formally, we represent an ϵ-NFA A by $A = (Q, \Sigma, \delta, q_0, F)$, where all components have their same interpretation as for an NFA, except that δ is now a function that takes as arguments:

1. A state in Q
2. A member of $\Sigma \cup \{\epsilon\}$, that is, either an input symbol, or the symbol ϵ. We require that ϵ, the symbol for the empty string, cannot be a member of the alphabet Σ, so no confusion results.

Elimination of ε-Transition

Given any ϵ-NFA E, we can find a DFA D that accepts the same language as E. The construction we use is very close to the subset construction, as the states of D are subsets of the states of E. The only difference is that we must incorporate ϵ-transitions of E, which we do through the mechanism of the ϵ-closure.

Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. Then the equivalent DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

is defined as follows:

1. Q_D is the set of subsets of Q_E. More precisely, we shall find that all accessible states of D are *ϵ-closed* subsets of Q_E, that is, sets $S \subseteq Q_E$ such that $S = \text{ECLOSE}(S)$. Put another way, the ϵ-closed sets of states S are those such that any ϵ-transition out of one of the states in S leads to a state that is also in S. Note that $\emptyset$ is an ϵ-closed set.

2. $q_D = \text{ECLOSE}(q_0)$; that is, we get the start state of D by closing the set consisting of only the start state of E. Note that this rule differs from the original subset construction, where the start state of the constructed automaton was just the set containing the start state of the given NFA.

3. F_D is those sets of states that contain at least one accepting state of E. That is, $F_D = \{S \mid S \text{ is in } Q_D \text{ and } S \cap F_E \neq \emptyset\}$.

Theorem 1:

A language L is accepted by some DFA iff L is accepted by some NFA.

Let language $L \subseteq \Sigma^*$, and suppose L is accepted by NFA $N = (\Sigma, Q, q_0, F, \delta)$. There exists a DFA $D= (\Sigma, Q', q'_0, F', \delta')$ that also accepts L. (L(N) = L(D)).

Proof:

By allowing each state in the DFA D to represent a set of states in the NFA N, we are able to prove through induction that D is equivalent to N. Before we begin the proof, let's define the parameters of D:

1. Q' is equal to the powerset of Q, $Q' = 2^Q$
2. $q'_0 = \{q_0\}$
3. F' is the set of states in Q' that contain any element of F, $F' = \{q \in Q' | q \cap F \neq \emptyset\}$
4. δ' is the transition function for D. $\delta'(q, a) = \bigcup_{p \in q} \delta(p, a)$ for $q \in Q'$ and $a \in \Sigma$.

Remember that each state in the set of states Q' in D is a set of states itself from Q in N. For each state p and state q in Q' of D (p is a single state from Q), determine the transition $\delta(p,a)$. $\delta(q,a)$ is the union of all $\delta(p,a)$.

Now we will prove that $\hat{\delta}'(q'_0, x) = \hat{\delta}(q_0, x)$ for every *x*. ie, L(D) = L(N)

Basis Step

Let *x* be the empty string ε.

$$\begin{aligned}
\hat{\delta}'(q'_0, x) &= \hat{\delta}'(q'_0, \epsilon) \\
&= q'_0 \\
&= \{q_0\} \\
&= \hat{\delta}(q_0, \epsilon) \\
&= \hat{\delta}(q_0, x)
\end{aligned}$$

Inductive Hypothesis: Assume that for any *y* with $|y| \geq 0$, . $\hat{\delta}'(q'_0, y) = \hat{\delta}(q_0, y)$.

If we let $n = |y|$, then we need to prove that for a string z with $|z| = n + 1$,then

$\hat{\delta}'(q_0', z) = \hat{\delta}(q_0, z)$.

We can represent the string z as a concatenation of string y ($|y| = n$) and symbol a from the alphabet Σ ($a \in \Sigma$). So, $z = ya$.

DFA D accepts a string x iff $\hat{\delta}'(q_0', x) \in F'$. From the above it follows that D accepts x iff $\hat{\delta}(q_0, x) \cap F \neq \emptyset$

$$\begin{aligned}
\hat{\delta}'(q_0', z) &= \hat{\delta}'(q_0', ya) \\
&= \delta'(\hat{\delta}(q_0', y), a) \\
&= \delta'(\hat{\delta}(q_0, y), a) \qquad \text{(by assumption)} \\
&= \bigcup_{p \in \hat{\delta}(q_0, y)} \hat{\delta}(p, a) \qquad \text{(by definition of } \delta'\text{)} \\
&= \hat{\delta}(q_0, ay) \\
&= \hat{\delta}(q_0, z)
\end{aligned}$$

DFA D accepts a string x iff $\hat{\delta}'(q_0', x) \in F'$. From the above it follows that D accepts x iff $\hat{\delta}(q_0, x) \cap F \neq \emptyset$.

So a string is accepted by DFA D if, and only if, it is accepted by NFA N.

Steps for Converting NFA to DFA:

Step 1: Initially Q' = ϕ

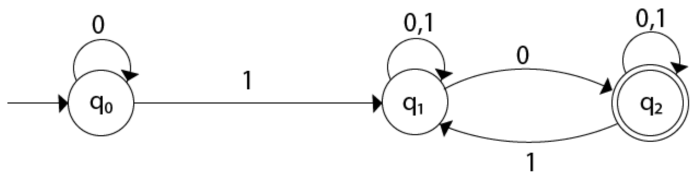
Step 2: Add q0 of NFA to Q'. Then find the transitions from this start state.

Step 3: In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.

Step 4: In DFA, the final state will be all the states which contain F(final states of NFA)

Problem 2:

Convert the given NFA to DFA.



Solution: For the given transition diagram we will first construct the transition table.

s \ inpu	0	1
→ q_0	q_0	q_1
q_1	$\{q_1, q_2\}$	Q_1
q_2	q_2	$\{q_1, q_2\}$

Now we will obtain δ' transition for state q_0.

1. $\delta'([q_0], 0) = [q_0]$
2. $\delta'([q_0], 1) = [q_1]$

The δ' transition for state q1 is obtained as:

1. $\delta'([q_1], 0) = [q_1, q_2]$ (**new** state generated)
2. $\delta'([q_1], 1) = [q_1]$

The δ' transition for state q_2 is obtained as:

1. $\delta'([q_2], 0) = [q_2]$
2. $\delta'([q_2], 1) = [q_1, q_2]$

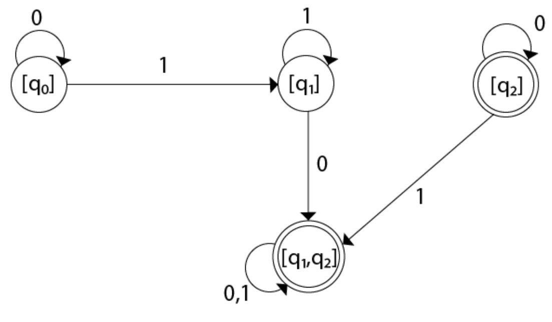
Now we will obtain δ' transition on $[q_1, q_2]$.

1. $\delta'([q_1, q_2], 0) = \delta(q_1, 0) \cup \delta(q_2, 0)$
   $= \{q_1, q_2\} \cup \{q_2\}$
   $= [q_1, q_2]$
2. $\delta'([q1, q2], 1) = \delta(q_1, 1) \cup \delta(q_2, 1)$
   $= \{q_1\} \cup \{q_1, q_2\}$
   $= \{q_1, q_2\}$
   $= [q_1, q_2]$

The state $[q_1, q_2]$ is the final state as well because it contains a final state q_2. The transition table for the constructed DFA will be:

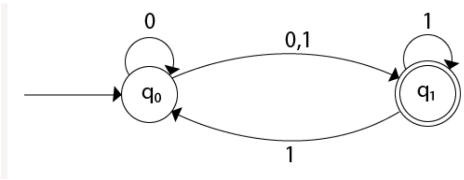
s \ input	0	1
→ $[q_0]$	$[q_0]$	$[q_1]$
$[q_1]$	$[q_1,q_2]$	$[q_1,q_2]$
$*[q_2]$	$[q_1,q_2]$	$[q_1,q_2]$

The Transition diagram will be:



The state q_2 can be eliminated because q_2 is an unreachable state.

Problem 3: Convert the given NFA to DFA.



Solution: For the given transition diagram we will first construct the transition table.

s \ input	0	1
→ q_0	$\{q_0,q_1\}$	q_1
q_1	ϕ	$\{q_0,q_1\}$

Now we will obtain δ' transition for state q_0.

1. $\delta'([q0], 0) = \{q0, q1\} = [q0, q1]$ (**new** state generated)
2. $\delta'([q0], 1) = \{q1\} = [q1]$

The δ' transition for state q_1 is obtained as:

1. $\delta'([q_1], 0) = \phi$
2. $\delta'([q_1], 1) = [q_0, q_1]$

Now we will obtain δ' transition on $[q_0, q_1]$.

1. $\delta'([q_0, q_1], 0) = \delta(q_0, 0) \cup \delta(q_1, 0)$

$$= \{q_0, q_1\} \cup \phi$$
$$= \{q_0, q_1\}$$
$$= [q_0, q_1]$$

Similarly,

1. $\delta'([q_0, q_1], 1) = \delta(q_0, 1) \cup \delta(q_1, 1)$
2. $= \{q_1\} \cup \{q_0, q_1\}$
3. $= \{q_0, q_1\}$
4. $= [q_0, q_1]$

As in the given NFA, q_1 is a final state, then in DFA wherever, q_1 exists that state becomes a final state.

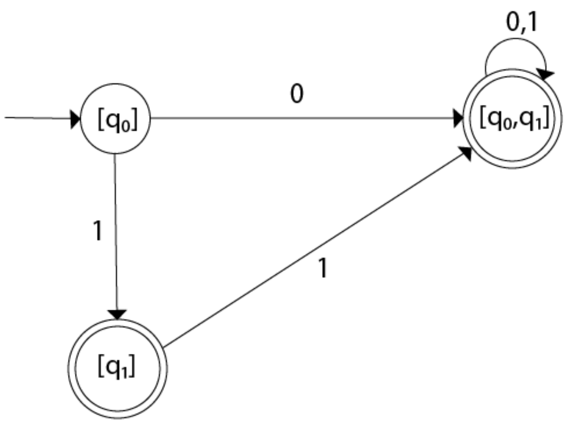
Hence in the DFA, final states are $[q_1]$ and $[q_0, q_1]$.

Therefore set of final states $F = \{[q_1], [q_0, q_1]\}$.

The transition table for the constructed DFA will be:

s \ input	0	1
→ $[q_0]$	$[q_0,q_1]$	$[q_1]$
$*[q_1]$	ϕ	$[q_0,q_1]$
$*[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_1]$

The Transition diagram will be:



Conversion from NFA with ε to DFA

Non-deterministic finite automata(NFA) is a finite automata where for some cases when a specific input is given to the current state, the machine goes to multiple states or more than 1 states. It can contain ε move. It can be represented as M = { Q, ∑, δ, q0, F}.

Where

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: final state
5. δ: Transition function

NFA with ε move: If any FA contains ε transaction or move, the finite automata is called NFA with **ε** move.

Steps for converting NFA with ε to DFA:

Step 1: We will take the ε-closure for the starting state of NFA as a starting state of DFA.

Step 2: Find the states for each input symbol that can be traversed from the present. That means the union of transition value and their closures for each state of NFA present in the current state of DFA.

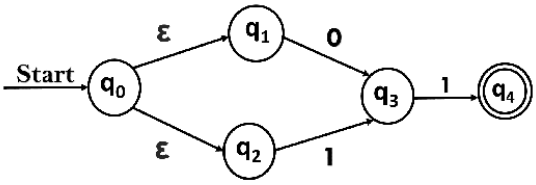
Step 3: If we found a new state, take it as current state and repeat step 2.

Step 4: Repeat Step 2 and Step 3 until there is no new state present in the transition table of DFA.

Step 5: Mark the states of DFA as a final state which contains the final state of NFA.

Problem 4:

Convert the NFA with ε into its equivalent DFA.



Solution:

Let us obtain ε-closure of each state.

1. ε-closure $\{q_0\} = \{q_0, q_1, q_2\}$
2. ε-closure $\{q_1\} = \{q_1\}$
3. ε-closure $\{q_2\} = \{q_2\}$
4. ε-closure $\{q_3\} = \{q_3\}$
5. ε-closure $\{q_4\} = \{q_4\}$

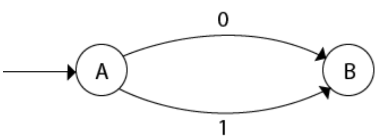
Now, let ε-closure $\{q_0\} = \{q_0, q_1, q_2\}$ be named as A.

Hence

$\delta'(A, 0) =$ ε-closure $\{\delta((q_0, q_1, q_2), 0)\}$
$=$ ε-closure $\{\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0)\}$
$=$ ε-closure $\{q_3\}$
$= \{q_3\}$ **call it as state B**.

$\delta'(A, 1) =$ ε-closure $\{\delta((q_0, q_1, q_2), 1)\}$
$=$ ε-closure $\{\delta((q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1)\}$
$=$ ε-closure $\{q_3\}$
$= \{q_3\} =$ B.

The partial DFA will be



Now,

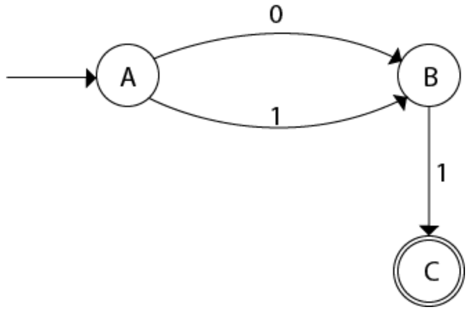
$\delta'(B, 0) = \varepsilon\text{-closure } \{\delta(q_3, 0)\}$
$= \phi$

$\delta'(B, 1) = \varepsilon\text{-closure } \{\delta(q_3, 1)\} = \varepsilon\text{-closure } \{q_4\} = \{q_4\}$ **i.e. state C**

For state C:

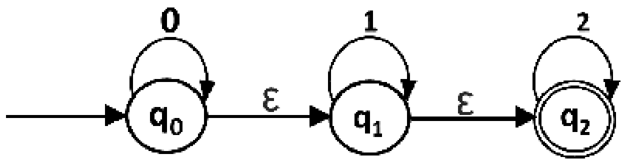
1. $\delta'(C, 0) = \varepsilon\text{-closure } \{\delta(q_4, 0)\} = \phi$
2. $\delta'(C, 1) = \varepsilon\text{-closure } \{\delta(q_4, 1)\} = \phi$

The DFA will be,



Problem 5:

Convert the given NFA into its equivalent DFA.



Solution: Let us obtain the ε-closure of each state.

1. $\varepsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$
2. $\varepsilon\text{-closure}(q_1) = \{q_1, q_2\}$
3. $\varepsilon\text{-closure}(q_2) = \{q_2\}$

Now we will obtain δ' transition. Let $\varepsilon\text{-closure}(q0) = \{q_0, q_1, q_2\}$ call it as **state A**.

$\delta'(A, 0) = \varepsilon\text{-closure}\{\delta((q_0, q_1, q_2), 0)\}$
$= \varepsilon\text{-closure}\{\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0)\}$
$= \varepsilon\text{-closure}\{q_0\}$
$= \{q_0, q_1, q_2\}$

$\delta'(A, 1) = \varepsilon\text{-closure}\{\delta((q_0, q_1, q_2), 1)\}$
$= \varepsilon\text{-closure}\{\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1)\}$
$= \varepsilon\text{-closure}\{q_1\}$

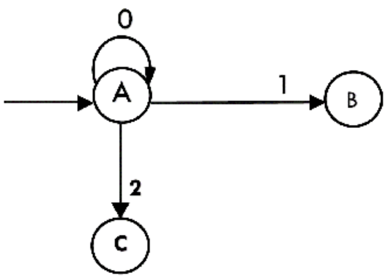
$= \{q_1, q_2\}$ **call it as state B**

$\delta'(A, 2) = \varepsilon\text{-closure}\{\delta((q_0, q_1, q_2), 2)\}$
$= \varepsilon\text{-closure}\{\delta(q_0, 2) \cup \delta(q_1, 2) \cup \delta(q_2, 2)\}$
$= \varepsilon\text{-closure}\{q_2\}$
$= \{q_2\}$ **call it state C**

Thus we have obtained

1. $\delta'(A, 0) = A$
2. $\delta'(A, 1) = B$
3. $\delta'(A, 2) = C$

The partial DFA will be:



Now we will find the transitions on states B and C for each input.

Hence

$\delta'(B, 0) = \varepsilon\text{-closure}\{\delta((q_1, q_2), 0)\}$
$= \varepsilon\text{-closure}\{\delta(q_1, 0) \cup \delta(q_2, 0)\}$
$= \varepsilon\text{-closure}\{\phi\}$
$= \phi$

$\delta'(B, 1) = \varepsilon\text{-closure}\{\delta((q_1, q_2), 1)\}$
$= \varepsilon\text{-closure}\{\delta(q_1, 1) \cup \delta(q_2, 1)\}$
$= \varepsilon\text{-closure}\{q_1\}$
$= \{q_1, q2\}$ **i.e. state B itself**

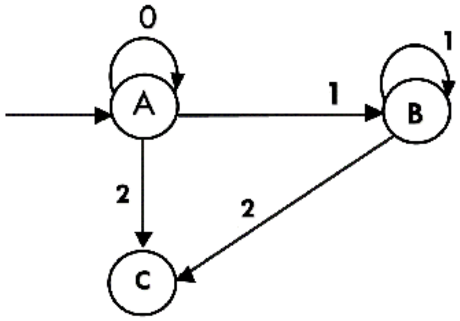
$\delta'(B, 2) = \varepsilon\text{-closure}\{\delta((q_1, q_2), 2)\}$
$= \varepsilon\text{-closure}\{\delta(q_1, 2) \cup \delta(q_2, 2)\}$
$= \varepsilon\text{-closure}\{q_2\}$
$= \{q_2\}$ **i.e. state C itself**

Thus we have obtained

1. $\delta'(B, 0) = \phi$

2. $\delta'(B, 1) = B$
3. $\delta'(B, 2) = C$

The partial transition diagram will be



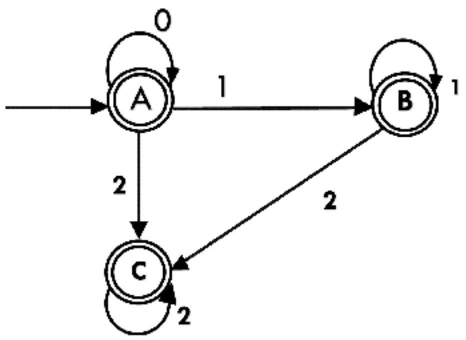
Now we will obtain transitions for C:

$$\delta'(C, 0) = \varepsilon\text{-closure}\{\delta(q_2, 0)\}$$
$$= \varepsilon\text{-closure}\{\phi\}$$
$$= \phi$$

$$\delta'(C, 1) = \varepsilon\text{-closure}\{\delta(q_2, 1)\}$$
$$= \varepsilon\text{-closure}\{\phi\}$$
$$= \phi$$

$$\delta'(C, 2) = \varepsilon\text{-closure}\{\delta(q_2, 2)\}$$
$$= \{q_2\}$$

Hence the DFA is



As A = $\{q_0, q_1, q_2\}$ in which final state q_2 lies hence A is final state. B = $\{q_1, q_2\}$ in which the state q_2 lies hence B is also final state. C = $\{q_2\}$, the state q_2 lies hence C is also a final state.

Regular Expressions

- 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.

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.
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:

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

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

Problem 6:

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

R = a*

That is Kleen closure of a.

Problem 7:

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:

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.

Problem 8:

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:
R = 1 (0+1)* 0

Problem 9:
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: R = a b* b

Problem 10:
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:R = a* b* c*

Conversion of Regular Expression into DFA

Theorem 1
Every language de fined by a regular expression is also defined by a finite automaton

Steps to follow for the conversion
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.

Properties of Regular Sets

Property 1. The union of two regular set is regular.

Property 2. The intersection of two regular set is regular.

Property 3. The complement of a regular set is regular

Property 4. The difference of two regular set is regular.

Property 5. The reversal of a regular set is regular.

Property 6. The closure of a regular set is regular

Property 7. The concatenation of two regular sets is regular.

Identities Related to Regular Expressions

Given R, P, L, Q as regular expressions, the following identities hold −

- ∅* = ε
- ε* = ε
- RR* = R*R
- R*R* = R*
- (R*)* = R*
- RR* = R*R
- (PQ)*P =P(QP)*
- (a+b)* = (a*b*)* = (a*+b*)* = (a+b*)* = a*(ba*)*
- R + ∅ = ∅ + R = R (The identity for union)
- R ε = ε R = R (The identity for concatenation)
- ∅ L = L ∅ = ∅ (The annihilator for concatenation)
- R + R = R (Idempotent law)
- L (M + N) = LM + LN (Left distributive law)
- (M + N) L = ML + NL (Right distributive law)
- ε + RR* = ε + R*R = R*

Pumping Lemma

Let L be a regular language. Then there exists a constant **'k'** such that for every string **w** in **L** −

|w| ≥ k

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

- $|y| > 0$
- $|xy| \leq k$
- For all $i \geq 0$, the string xy^iz is also in L.

Applications 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 Pumping Lemma.
- If **L** does not satisfy Pumping Lemma, it is non-regular.

Method to prove that a language L is not regular

- At first, we have to assume that **L** is regular.
- So, the pumping lemma should hold for **L**.
- Use the pumping lemma to obtain a contradiction −
    - Select **w** such that **|w| ≥ k**
    - Select **y** such that **|y| ≥ 1**

- Select **x** such that **|xy| ≤ k**
- Assign the remaining string to **z.**
- Select **i** such that the resulting string is not in **L.**

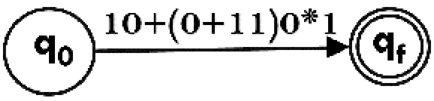
Hence L is not regular.

Problem11:

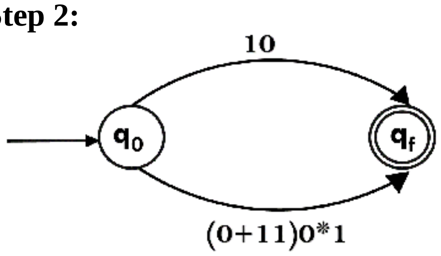
Design a FA from given regular expression 10 + (0 + 11)0* 1.

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

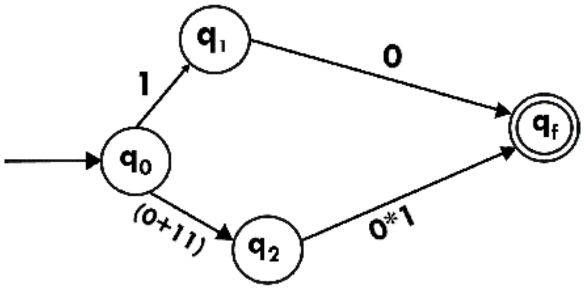
Step 1:



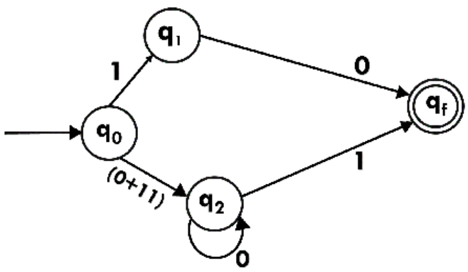
Step 2:



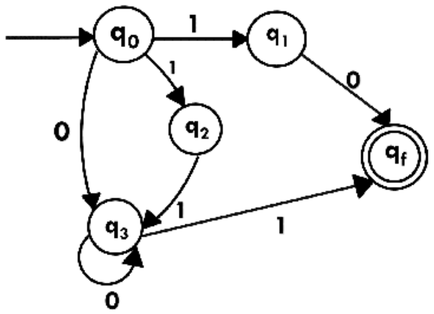
Step 3:



Step 4:



Step 5:



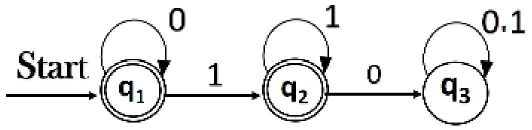
Now we have got NFA without ε. Now we will convert it into required DFA for that, we will first write a transition table for this NFA.

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

Problem 12: Construct the regular expression for the given DFA



Solution:

Let us write down the equations

$q_1 = q_1\ 0 + \varepsilon$

Since q_1 is the start state, so ε will be added, and the input 0 is coming to q_1 from q_1 hence we write **State = source state of input × input coming to it.**

Similarly,
$q_2 = q_1 1 + q_2 1$
$q_3 = q_2 0 + q_3 (0+1)$

Since the final states are q_1 and q_2, we are interested in solving q_1 and q_2 only.
Let us see q1 first
q1 = q1 0 + ε

We can re-write it as
$q_1 = ε + q_1 0$
Which is similar to R = Q + RP, and gets reduced to R = OP*.
Assuming R = q_1, Q = ε, P = 0
We get
q1 = ε.(0)*
q1 = 0* (ε.R*= R*)
Substituting the value into q_2, we will get
q2 = 0* 1 + q_2 1
q2 = 0* 1 (1)* (R = Q + RP → Q P*)

The regular expression is given by
$r = q_1 + q_2$
$= 0^* + 0^* 1.1^*$
$r = 0^* + 0^* 1^+ \quad (1.1^* = 1^+)$

Problem 12: Prove that **L = {a^ib^i | i ≥ 0}** is not regular.

Solution −

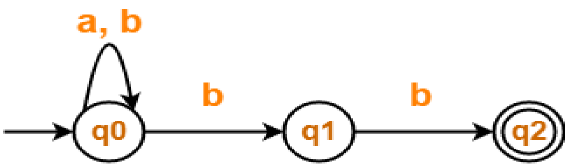
- At first, we assume that **L** is regular and n is the number of states.
- Let w = a^nb^n. Thus |w| = 2n ≥ n.
- By pumping lemma, let w = xyz, where |xy| ≤ n.
- Let x = a^p, y = a^q, and z = a^rb^n, where p + q + r = n, p ≠ 0, q ≠ 0, r ≠ 0. Thus |y| ≠ 0.
- Let k = 2. Then $xy^2z = a^pa^{2q}a^rb^n$.
- Powers of a = (p + 2q + r) = (p + q + r) + q = n + q
- Hence, $xy^2z = a^{n+q} b^n$. Since q ≠ 0, xy^2z is not of the form a^nb^n.
- Thus, xy^2z is not in L. Hence L is not regular.

Problem for Practice:

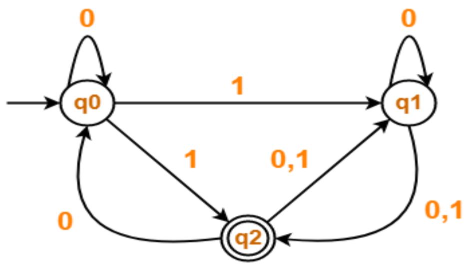
1. Convert to a DFA the following NFA

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\emptyset$
$*s$	$\{s\}$	$\{s\}$

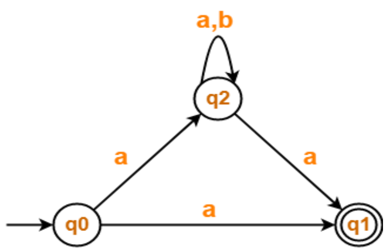
2. Convert the following Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA)-



3. Convert the following Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA)-



4. Convert the following Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA)-



5. Problem Consider the following NFA

	ϵ	a	b	c
$\rightarrow p$	$\emptyset$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\emptyset$
$*r$	$\{q\}$	$\{r\}$	$\emptyset$	$\{p\}$

a) Compute the ε- closure of each state.
b) Give all the strings of length three or less accepted by the automaton.
c) Convert the automaton to a DFA.

SCHOOL OF SCIENCE AND HUMANITIES

DEPARTMENT OF MATHEMATICS

UNIT – III - Formal Language and Automata – SMT1404

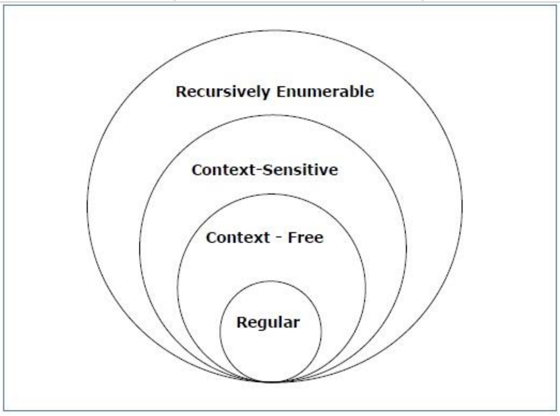
Introduction

In the literary sense of the term, grammars denote syntactical rules for conversation in natural languages. Linguistics have attempted to define grammars since the inception of natural languages like English, Sanskrit, Mandarin, etc.

The theory of formal languages finds its applicability extensively in the fields of Computer Science. Panini gave a grammar for the Sanskrit language. In 1959, Noam Chomsky tried to give a mathematical definition for grammar. The motivation was to give a formal definition for grammar for English sentences. According to Noam Chomosky, there are four types of grammars called Type 0, Type 1, Type 2, and Type 3..

The following table shows how they differ from each other :

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton



Formal Definition of a Grammar

A grammar **G** can be formally written as a 4-tuple (N, T, S, P) where −

- **N** or **V_N** is a set of variables or non-terminal symbols.
- **T** or ∑ is a set of Terminal symbols.
- **S** is a special variable called the Start symbol, $S \in N$
- **P** is Production rules for Terminals and Non-terminals. A production rule has the form $\alpha \rightarrow \beta$, where α and β are strings on $V_N \cup \sum$ and least one symbol of α belongs to V_N.

A Grammar is mainly composed of two basic elements-

1. Terminal symbols
2. Non-terminal symbols

1. **Terminal Symbols-**
   - Terminal symbols are those which are the constituents of the sentence generated using a grammar.
   - Terminal symbols are denoted by using small case letters such as a, b, c etc.
2. **Non-Terminal Symbols-**
   - Non-Terminal symbols are those which take part in the generation of the sentence but are not part of it.
   - Non-Terminal symbols are also called as auxiliary symbols or variables.
   - Non-Terminal symbols are denoted by using capital letters such as A, B, C etc.

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of $\boldsymbol{\alpha} \rightarrow \boldsymbol{\beta}$ where $\boldsymbol{\alpha}$ is a string of terminals and non-terminals with at least one non-terminal and $\boldsymbol{\alpha}$ cannot be null. $\boldsymbol{\beta}$ is a string of terminals and non-terminals.

Type-0 grammars include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. These languages are also known as the *recursively enumerable* or *Turing-recognizable* languages. Note that this is different from the recursive languages, which can be *decided* by an always-halting Turing machine.

Example

S → ACaB
Bc → acB
CB → DB
aD → Db

Type - 1 Grammar (CSG): Type-1 grammars generate context-sensitive languages. The productions must be in the form $\alpha\ A\ \beta \rightarrow \alpha\ \gamma\ \beta$, where $A \in N$ (Non-terminal) and $\alpha, \beta, \gamma \in (T \cup N)^*$ (Strings of terminals and non-terminals). The strings α and β may be empty, but γ must be non-empty. The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

AB → AbBc
A → bcA
B → b

Type - 2 Grammar(CFG): Type-2 grammars generate context-free languages.
The productions must be in the form $A \rightarrow \gamma$,where $A \in N$ (Non terminal),
and $\gamma \in (T \cup N)^*$ (String of terminals and non-terminals).

Example

S → X a
X → a
X → aX
X → abc
X → ε

Type - 3 Grammar: (RE) Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

Type-3 grammars generate the regular languages. Such a grammar restricts its rules to a single non terminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single non terminal (right regular).

Alternatively, the right-hand side of the grammar can consist of a single terminal, possibly preceded by a single non terminal (left regular). These generate the same languages. However, if left-regular rules and right-regular rules are combined, the language need no longer be regular. These languages are exactly all languages that can be decided by a finite state automaton.

Additionally, this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

(i.e)The productions must be in the form **X → a or X → aY,**where **X, Y ∈ N** (Non terminal) and **a ∈ T** (Terminal). The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

X → ε
X → a | aY
Y → b

Example 1:

Grammar G1 =({S, A, B}, {a, b}, S, {S → AB, A → a, B → b})

Here,

- **S, A,** and **B** are Non-terminal symbols;
- **a** and **b** are Terminal symbols
- **S** is the Start symbol, S ∈ N
- Productions, **P : S → AB, A → a, B → b**

Example 2:

Grammar G2 =(({S, A}, {a, b}, S,{S → aAb, aA → aaAb, A → ε })

Here,

- **S** and **A** are Non-terminal symbols.
- **a** and **b** are Terminal symbols.
- **ε** is an empty string.

- **S** is the Start symbol, S ∈ N
- Production **P : S → aAb, aA → aaAb, A → ε**

Derivations from a Grammar

Strings may be derived from other strings using the productions in a grammar. If a grammar **G** has a production **α → β**, we can say that **x α y** derives **x β y** in **G**. This derivation is written as: $x\,\alpha\,y \Rightarrow_G x\,\beta\,y$.

Language generated by a Grammar

The set of all strings that can be derived from a grammar is said to be the language generated from that grammar. A language generated by a grammar **G** is a subset formally defined by $L(G)=\{W|W \in \sum^*, S \Rightarrow_G \mathbf{W}\}$

Equivalent Grammars-

Two grammars are said to be equivalent if they generate the same languages.

Example 3:

Consider the following two grammars **Grammar G_1-with the productions** S → aSb / ε

Grammar G_2-S → aAb / ε.

Both these grammars generate the same language L = { a^nb^n , n>=0 }.

Thus, L(G1) = L(G2).

Problem 1:

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

Solution:

As we know the regular expression for the above language is 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.

S⇒ aS⇒ aaS ⇒ aaaS⇒ aaaaS⇒ aaaaaS⇒ 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.

Problem 2: Derive the string aaabbb from the grammar G2 = ({S, A}, {a, b}, S, {S → aAb, aA → aaAb, A → ε })

the string that can be derived here :−

S ⇒ aAb using production S → aAb

⇒ aaAbb using production aA → aAb

⇒ aaaAbbb using production aA → aAb

⇒ aaabbb using production A → ε

Problem 3:

Find the language generated by the grammar G=: ({S, A, B}, {a, b} , {S → AB, A → aA|a, B → bB|b})

Solution:

The language generated by this grammar −

L(G) = {ab, a^2b, ab^2, a^2b^2, ………}

= {a^m b^n | m ≥ 1 and n ≥ 1}.

Problem 4: Construct the grammar generating the language

L (G)={a^m b^n | m ≥ 0 and n > 0}.

Solution:

Since L(G) = {a^m b^n | m ≥ 0 and n > 0}

the set of strings accepted can be rewritten as −

L(G) = {b, ab,bb, aab, abb, …….}

Here, the start symbol has to take at least one 'b' preceded by any number of 'a' including null.

To accept the string set {b, ab, bb, aab, abb, …….}, we have taken the productions −

S → aS , S → B, B → b and B → bB

S → B → b (Accepted)

S → B → bB → bb (Accepted)

S → aS → aB → ab (Accepted)

S → aS → aaS → aaB → aab(Accepted)

S → aS → aB → abB → abb (Accepted)

Thus, we can prove every single string in L(G) is accepted by the language generated by the production set.

Hence the grammar is G: ({S, A, B}, {a, b}, S, { S → aS | B , B → b | bB })

Problem 5 Find the grammar generating the language L (G) = {a^m b^n | m > 0 and n ≥ 0}.

Solution:

Since L(G) = {a^m b^n | m > 0 and n ≥ 0}, the set of strings accepted can be rewritten as −

L(G) = {a, aa, ab, aaa, aab ,abb, …….}

Here, the start symbol has to take at least one 'a' followed by any number of 'b' including null.

To accept the string set {a, aa, ab, aaa, aab, abb, …….}, we have taken the productions −

S → aA, A → aA , A → B, B → bB ,B → λ

S → aA → aB → aλ → a (Accepted)

S → aA → aaA → aaB → aaλ → aa (Accepted)

$S \rightarrow aA \rightarrow aB \rightarrow abB \rightarrow ab\lambda \rightarrow ab$ (Accepted)

$S \rightarrow aA \rightarrow aaA \rightarrow aaaA \rightarrow aaaB \rightarrow aaa\lambda \rightarrow aaa$ (Accepted)

$S \rightarrow aA \rightarrow aaA \rightarrow aaB \rightarrow aabB \rightarrow aab\lambda \rightarrow aab$ (Accepted)

$S \rightarrow aA \rightarrow aB \rightarrow abB \rightarrow abbB \rightarrow abb\lambda \rightarrow abb$ (Accepted)

Thus, we can prove every single string in L(G) is accepted by the language generated by the production set.

Hence the grammar is G: ({S, A, B}, {a, b}, S, {$S \rightarrow aA$, $A \rightarrow aA \mid B$, $B \rightarrow \lambda \mid bB$ })

Problem 6: Construct a CFG for the regular expression (0+1)*
Solution:
The CFG can be given by,
1. Production rule (P):
2. $S \rightarrow 0S \mid 1S$
3. $S \rightarrow \varepsilon$

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 $S \rightarrow \varepsilon$.

Problem 7:
Construct a CFG for a language L = {wcw^R | where w ϵ (a, b)*}.
Solution:
The string that can be generated for a given language is
{aacaa, bcb, abcba, bacab, abbcbba, ...}
The grammar could be:
1. $S \rightarrow aSa$ rule 1
2. $S \rightarrow bSb$ rule 2
3. $S \rightarrow c$ rule 3

Now if we want to derive a string "abbcbba", we can start with start symbols.
1. $S \rightarrow aSa$
2. $S \rightarrow abSba$ from rule 2
3. $S \rightarrow abbSbba$ from rule 2
4. $S \rightarrow abbcbba$ from rule 3

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

Problem 8:
Construct a CFG for the language L = a^nb^{2n} where $n \geq 1$.
Solution:
The string that can be generated for a given language is {abb, aabbbb, aaabbbbbb....}.
The grammar could be:
1. $S \rightarrow aSbb \mid abb$
2. Now if we want to derive a string "aabbbb", we can start with start symbols.
3. $S \rightarrow aSbb$
4. $S \rightarrow aabbbb$

Problem 9: Find the context-Free Grammar for the following Languages

(a) $L = \{a^nb^m : n \leq m + 3\}$

(b) $L = \{a^n b^m : n \neq m - 1\}$

(c) $L = \{a^n b^m : n \neq 2m\}$

(d) $L = \{a^n b^m : n \neq 2m\}$

(e) $\{w \in \{a, b\}^* : n_a(w) \neq n_b(w)\}$

Solution:

The production rules are given for the languages generated

(a) $L = \{a^n b^m : n \leq m + 3\}$

$$\begin{aligned} \mathrm{S} &\rightarrow \mathrm{aSb} \mid \mathrm{T} \\ \mathrm{T} &\rightarrow \mathrm{Tb} \mid \mathrm{a} \mid \mathrm{aa} \mid \mathrm{aaa} \mid \lambda \end{aligned}$$

(b) $L = \{a^n b^m : n \neq m - 1\}$

$$\begin{aligned} \mathrm{S} &\rightarrow \mathrm{AT} \mid \mathrm{TB} \\ \mathrm{T} &\rightarrow \mathrm{aTb} \mid \mathrm{b} \\ \mathrm{A} &\rightarrow \mathrm{Aa} \mid \mathrm{a} \\ \mathrm{B} &\rightarrow \mathrm{Bb} \mid \mathrm{b} \end{aligned}$$

(c) $L = \{a^n b^m : n \neq 2m\}$

$$\begin{aligned} \mathrm{S} &\rightarrow \mathrm{AT} \mid \mathrm{TB} \\ \mathrm{T} &\rightarrow \mathrm{aaTb} \mid \lambda \\ \mathrm{A} &\rightarrow \mathrm{Aa} \mid \mathrm{a} \\ \mathrm{B} &\rightarrow \mathrm{Bb} \mid \mathrm{b} \end{aligned}$$

(d) $L = \{a^n b^m : n \neq 2m\}$

$$\mathrm{S} \rightarrow \mathrm{aSbb} \mid \mathrm{aSbbb} \mid \lambda$$

(e) $\{w \in \{a, b\}^* : n_a(w) \neq n_b(w)\}$

$$\begin{aligned} \mathrm{S} &\rightarrow \mathrm{aSb} \mid \mathrm{bSa} \mid \mathrm{A} \mid \mathrm{B} \\ \mathrm{A} &\rightarrow \mathrm{Aa} \mid \mathrm{a} \end{aligned}$$

Problem for Practice:

1. Design context free grammars for the following languages

a) The set $\{0^n 1^n \mid n \geq 1\}$, that is, the set of all strings of one or more 0's followed by an equal number of 1's.

b) The set $\{a^i b^j c^k \mid i \neq j \text{ or } j \neq k\}$, that is, the set of strings of a's followed by b's followed by c's, such that there are either a different number of a's and b's or a different number of b's and c's, or both.

SCHOOL OF SCIENCE AND HUMANITIES

DEPARTMENT OF MATHEMATICS

UNIT – II - Formal Language and Automata – SMT1404

Introduction

Regular Languages :

The **collection of Regular Languages** over an alphabet Σ is defined recursively as follows:

(i) The empty language Ø, and the empty string language {ε} are regular languages.
(ii) For each $a \in \Sigma$ (a belongs to Σ), the singleton language $\{a\}$ is a regular language.
(iii) If A and B are regular languages, then $A \cup B$ (union), AB (concatenation), and A^* (Kleene star) are regular languages.
(iv) No other languages over Σ are regular.

Note: It is Type-3 Grammar.

Examples:

(i) All finite languages are regular.
(ii) In particular the empty string language {ε} = Ø* is regular.
(iii) Other typical examples include the language consisting of all strings over the alphabet {a, b} which contain an even number of a's.

Note:

Regular Languages can be represented by Regular Expressions.

Example: (i) L = Set of all strings over a, b that ends with bb.
L = {bb, abb, bbb, aabb, abbb, ababb,....}
Regular Expression: (a + b)*bb.
(ii) L = Set of all strings over 0, 1 with three consecutive 1's
L = {111,0111,1111,01110,01111,1110,11110,....}
Regular Expression: (0 + 1)*111(0+1)*.

Equivalent Formalism:

A regular language satisfies the following equivalent properties:
(i) it can be generated by a regular grammar
(ii) it is the language of a regular expression
(iii) it is the language accepted by a nondeterministic finite automaton (NFA) / deterministic finite automaton (DFA)

Closure Properties:

The regular languages are closed under various operations.
If the languages L and M are regular, then 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.

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.

$L \cap M$ = {st | s is in L and t is in M}

Kleen closure: If L is a regular language then its Kleen closure L_1* will also be a regular language.

Here Properties are proved by examples

Property 1: The union of two regular set is regular.

Proof:

Let us take two regular expressions $RE_1 = a(aa)^*$ and $RE_2 = (aa)^*$

So, $L_1 = \{a, aaa, aaaaa, ...\}$ (Strings of odd length excluding Null)

and $L_2 = \{ \varepsilon, aa, aaaa, aaaaaa, ...\}$ (Strings of even length including Null)

$L_1 \cup L_2 = \{ \varepsilon, a, aa, aaa, aaaa, aaaaa, aaaaaa, ...\}$

(Strings of all possible lengths including Null)

$RE (L_1 \cup L_2) = a^*$ (which is a regular expression itself)

Hence, the union of two regular languages is regular.

Property 2: The intersection of two regular set is regular.

Proof :

Let us take two regular expressions

$RE_1 = a(a^*)$ and $RE_2 = (aa)^*$

So, $L_1 = \{ a, aa, aaa, aaaa, ...\}$ (Strings of all possible lengths excluding Null)

$L_2 = \{ \varepsilon, aa, aaaa, aaaaaa, ...\}$ (Strings of even length including Null)

$L_1 \cap L_2 = \{ aa, aaaa, aaaaaa, ...\}$ (Strings of even length excluding Null)

$RE (L_1 \cap L_2) = aa(aa)^*$ which is a regular expression itself.

Hence, the intersection of two regular languages is regular.

Property 3. The complement of a regular set is regular.

Proof :

Let us take a regular expression

$RE = (aa)^*$

So, $L = \{\varepsilon, aa, aaaa, aaaaaa, ...\}$ (Strings of even length including Null)

Complement of **L** is all the strings that is not in **L**.

So, $L' = \{a, aaa, aaaaa, ...\}$ (Strings of odd length excluding Null)

$RE (L') = a(aa)^*$ which is a regular expression itself.

Hence, the result is proved.

Property 4: The difference of two regular set is regular.

Proof :

Let us take two regular expressions

$RE_1 = a (a^*)$ and $RE_2 = (aa)^*$

So, $L_1 = \{a, aa, aaa, aaaa, ...\}$ (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

$L_1 - L_2$ = {a, aaa, aaaaa, aaaaaaa, ...}

(Strings of all odd lengths excluding Null)

RE ($L_1 - L_2$) = a (aa)* which is a regular expression.

Property 5: The reversal of a regular set is regular.

Proof:

We have to prove $\mathbf{L^R}$ is also regular if **L** is a regular set.

Let, L = {01, 10, 11, 10}

RE (L) = 01 + 10 + 11 + 10

L^R = {10, 01, 11, 01}

RE (L^R) = 01 + 10 + 11 + 10 which is regular

Property 6: The closure of a regular set is regular.

Proof:

If L = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

i.e., RE (L) = a (aa)*

L* = {a, aa, aaa, aaaa , aaaaa,……………} (Strings of all lengths excluding Null)

RE (L*) = a (a)*

Property 7. :The concatenation of two regular sets is regular.

Proof :

Let RE_1 = (0+1)*0 and RE_2 = 01(0+1)*

Here, L_1 = {0, 00, 10, 000, 010, ...} (Set of strings ending in 0)

and L_2 = {01, 010,011,.....} (Set of strings beginning with 01)

Then, L_1 L_2 = {001,0010,0011,0001,00010,00011,1001,10010,.............}

Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)*001(0 + 1)*

Hence, it is proved.

String Homomorphism: (string substitution)
A string homomorphism (often referred to simply as a homomorphism in formal language theory) is a string substitution such that each character is replaced by a single string. That is, f(a)=s, where s is a string, for each character a.
Given a language L, the set f(L) is called the homomorphic image of L. The inverse homomorphic image of a string s is defined as $f^{-1}(s)$= { w / f(w)=s }.

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.
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.

Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

Representation Technique

- **Root vertex** − Must be labeled by the start symbol.
- **Vertex** − Labeled by a non-terminal symbol.
- **Leaves** − Labeled by a terminal symbol or ε.

If $S \rightarrow x_1x_2 \ldots\ldots x_n$ is a production rule in a CFG, then the parse tree / derivation tree will be as follows −

There are two different approaches to draw a derivation tree −

Bottom-up Approach −

- Starts from tree leaves
- Proceeds upward to the root which is the starting symbol **S**

Derivation or Yield of a Tree

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

Construction of Parse Tree

Let us fix on a grammar $G = (V, T, P, S)$. The *parse trees* for G are trees with the following conditions:

1. Each interior node is labeled by a variable in V.

2. Each leaf is labeled by either a variable, a terminal, or ϵ. However, if the leaf is labeled ϵ, then it must be the only child of its parent.

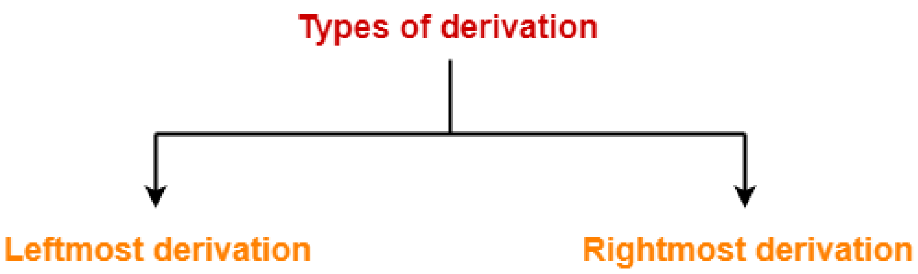
3. If an interior node is labeled A, and its children are labeled

$$X_1, X_2, \ldots, X_k$$

respectively, from the left, then $A \to X_1X_2 \cdots X_k$ is a production in P. Note that the only time one of the X's can be ϵ is if that is the label of the only child, and $A \to \epsilon$ is a production of G.

Parse Tree:

The process of deriving a string is called as **derivation**. The geometrical representation of a derivation is called as a **parse tree** or **derivation tree.**



Leftmost Derivation:

The process of deriving a string by expanding the leftmost non-terminal at each step is called as **leftmost derivation**. The geometrical representation of leftmost derivation is called as a **leftmost derivation tree**.

2. Rightmost Derivation;

The process of deriving a string by expanding the rightmost non-terminal at each step is called as **rightmost derivation**. The geometrical representation of rightmost derivation is called as a **rightmost derivation tree**.

Problem: 01

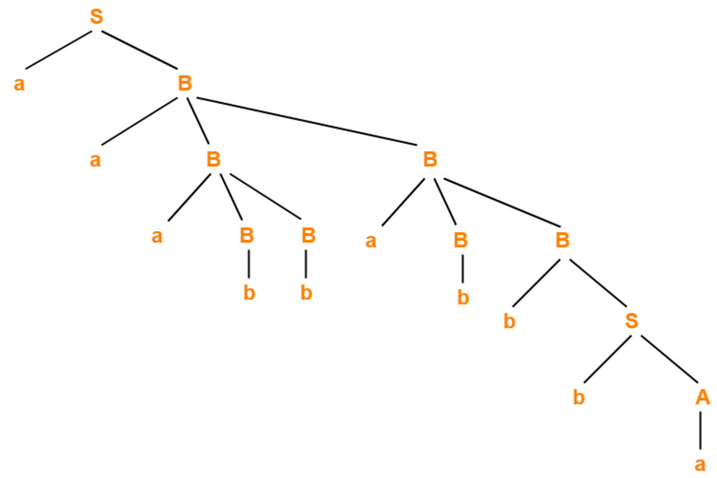
Find the left most derivation for the string w = aaabbabbba by the grammar G with the production B → bS / aBB / b (**Unambiguous Grammar**)

Solution:

Now, let us derive the string w using leftmost derivation.

Leftmost Derivation:

S ⇒a**B**
⇒ aa**B**B (Using B → aBB)
⇒ aaa**B**BB (Using B → aBB)
⇒aaab**B**B (Using B → b)
⇒ aaabb**B** (Using B → b)
⇒ aaabba**B**B (Using B → aBB)
⇒ aaabbab**B** (Using B → b)
⇒ aaabbabb**S** (Using B → bS)
⇒ aaabbabbb**A** (Using S → bA)
⇒aaabbabbba (Using A → a)



Leftmost Derivation Tree

Problem: 02

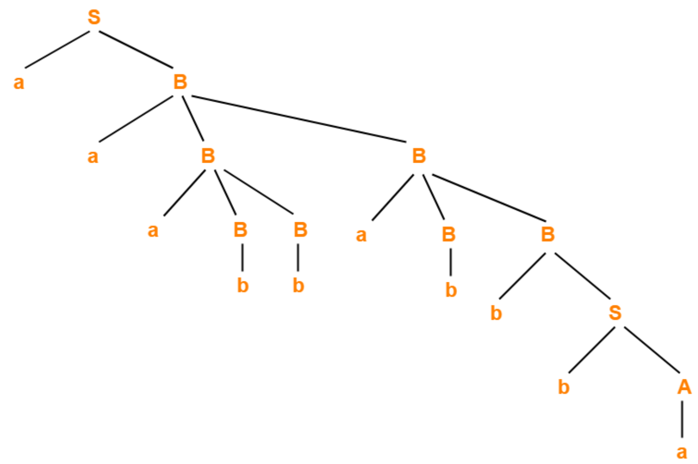
Find the right most derivation for the string w = aaabbabbba by the grammar S ⟶ aB / bA, S ⟶ aS / bAA / a, B ⟶ bS / aBB / b (**Unambiguous Grammar**)

Solution:

Now, let us derive the string w using rightmost derivation.

Rightmost Derivation:

S	⇒ a**B**	
	⇒ aaB**B**	(Using B ⟶ aBB)
	⇒ aaBaB**B**	(Using B ⟶ aBB)
	⇒ aaBaBb**S**	(Using B ⟶ bS)
	⇒ aaBaBbb**A**	(Using S ⟶ bA)
	⇒ aaBa**B**bba	(Using A ⟶ a)
	⇒ aa**B**abbba	(Using B ⟶ b)
	⇒ aaaB**B**abbba	(Using B ⟶ aBB)
	⇒aaa**B**babbba	(Using B ⟶ b)
	⇒ aaabbabbba	(Using B ⟶ b)



Rightmost Derivation Tree

Ambiguity in Grammar

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string. If the grammar is not ambiguous, then it is called unambiguous.

If the grammar has ambiguity, then it is not good for compiler construction. No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.

NOTE:

- For unambiguous grammars, there is a uniqueLeft-most derivation or Rightmost derivation.
- For ambiguous grammars, there is two or more Left-most derivation or Rightmost derivations.

Here,

- The given grammar was unambiguous.
- That is why, leftmost derivation and rightmost derivation represents the same parse tree.

Properties of Parse Tree:

- Root node of a parse tree is the start symbol of the grammar.
- Each leaf node of a parse tree represents a terminal symbol.
- Each interior node of a parse tree represents a non-terminal symbol.
- Parse tree is independent of the order in which the productions are used during derivations.

PROBLEMS BASED ON DERIVATIONS AND PARSE TREE-

Problem: 3

Consider the grammar-S → bB / aA; A → b / bS / aAA; B → a / aS / bBB

For the string w = bbaababa, find

1. Leftmost derivation
2. Rightmost derivation
3. Parse Tree

Solution:

1. Leftmost Derivation-

S ⇒ b**B**

⇒ bb**B**B (Using B → bBB)

⇒ bba**B** (Using B → a)

⇒ bbaa**S** (Using B → aS)

⇒ bbaab**B** (Using S → bB)

⇒ bbaaba**S** (Using B → aS)

⇒bbaabab**B** (Using S → bB)

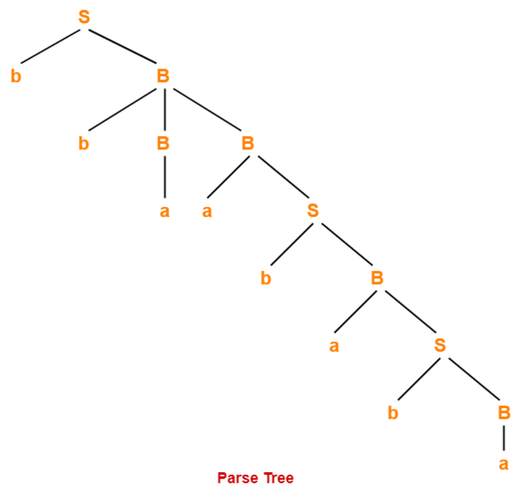
⇒ bbaababa (Using B → a)

2. Rightmost Derivation-

S ⇒ b**B**

⇒ bbB**B** (Using B → bBB)
⇒bbBa**S** (Using B → aS)
⇒ bbBab**B** (Using S → bB)
⇒bbBaba**S** (Using B → aS)
⇒ bbBabab**B** (Using S → bB)
⇒ bb**B**ababa (Using B → a)
⇒bbaababa (Using B → a)

3. Parse Tree:



Parse Tree

Whether we consider the leftmost derivation or rightmost derivation, we are getting a unique parse tree for both cases.

⇒The given grammar is unambiguous.

Problem: 4

Consider the grammar- S → A1B; A → 0A / ∈; B → 0B / 1B / ∈

For the string w = 00101, find-

1. Leftmost derivation
2. Rightmost derivation
3. Parse Tree

Solution:

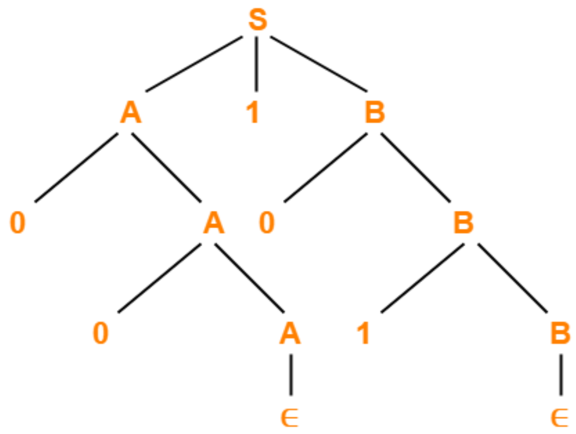
1. Leftmost Derivation:

S ⇒ **A**1B
⇒ 0**A**1B (Using A → 0A)
⇒ 00**A**1B (Using A → 0A)
⇒ 001**B** (Using A → ∈)
⇒ 0010**B** (Using B → 0B)
⇒ 00101**B** (Using B → 1B)
⇒00101 (Using B → ∈)

2. Rightmost Derivation-

S ⇒ A1**B**
⇒ A10B (Using B → 0B)
⇒ A101B (Using B → 1B)

⇒ A101 (Using B → ∈)
⇒0A101 (Using A → 0A)
⇒00A101 (Using A → 0A)
⇒ 00101 (Using A → ∈)



Parse Tree

Difference Between Ambiguous and Unambiguous Grammar:
Some of the important differences between ambiguous grammar and unambiguous grammar are-

Ambiguous Grammar	Unambiguous Grammar
A grammar is said to be ambiguous if for at least one string generated by it, it produces more than one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation	A grammar is said to be unambiguous if for all the strings generated by it, it produces exactly one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation
For ambiguous grammar, leftmost derivation and rightmost derivation represents different parse trees.	For unambiguous grammar, leftmost derivation and rightmost derivation represents the same parse tree.
Ambiguous grammar contains less number of non-terminals.	Unambiguous grammar contains more number of non-terminals.
For ambiguous grammar, length of parse tree is less.	For unambiguous grammar, length of parse tree is large.
Ambiguous grammar is faster than unambiguous grammar in the derivation of a tree. (Reason is above 2 points)	Unambiguous grammar is slower than ambiguous grammar in the derivation of a tree.

Example-	**Example-**
E → E + E / E x E / id (Ambiguous Grammar)	E → E + T / T T → T x F / F F → id (Unambiguous Grammar)

General Approach to Check Grammar Ambiguity:

To check whether a given grammar is ambiguous or not, we follow the following steps-

Step-01:

We try finding a string from the Language of Grammar such that for the string there exists more than one-

- parse tree
- or derivation tree
- or syntax tree
- or leftmost derivation
- or rightmost derivation

Step-02:

If there exists at least one such string, then the grammar is ambiguous otherwise unambiguous.

Unambiguous Grammar

A grammar can be unambiguous if the grammar does not contain ambiguity that means if it does not contain more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string.

To convert ambiguous grammar to unambiguous grammar, we will apply the following rules:

1. If the left associative operators (+, -, *, /) are used in the production rule, then apply left recursion in the production rule. Left recursion means that the leftmost symbol on the right side is the same as the non-terminal on the left side. For example, X → Xa.
2. If the right associative operates(^) is used in the production rule then apply right recursion in the production rule. Right recursion means that the rightmost symbol on the left side is the same as the non-terminal on the right side. For example, X → aX

Example:

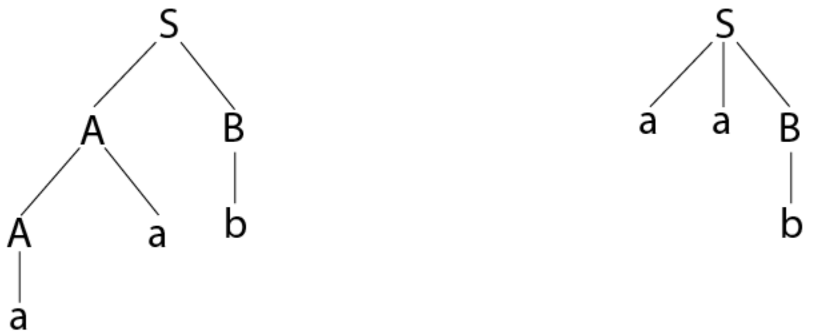
Consider a grammar G is given as follows:

1. S → AB | aaB
2. A → a | Aa
3. B → b

Determine whether the grammar G is ambiguous or not. If G is ambiguous, construct an unambiguous grammar equivalent to G.

Solution:

Let us derive the string "aab"



Parse tree 1 Parse tree 2

As there are two different parse tree for deriving the same string, the given grammar is ambiguous.

Unambiguous grammar will be:

1. S → AB
2. A → Aa | a
3. B → b .

PROBLEMS BASED ON CHECKING WHETHER GRAMMAR IS AMBIGUOUS-

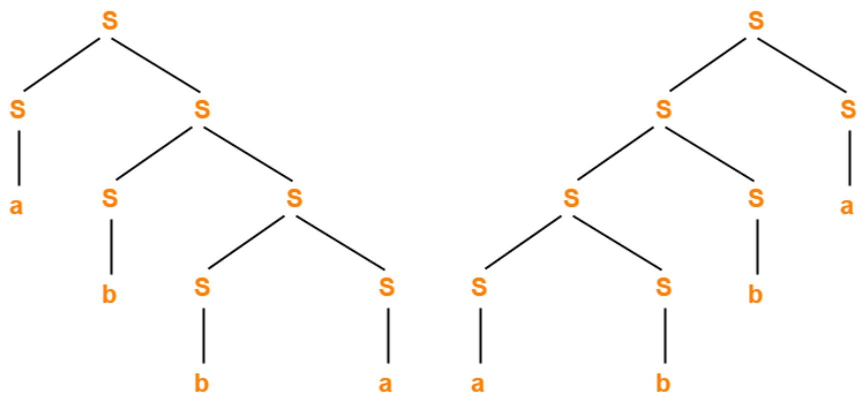
Problem:5

Check whether the given grammar is ambiguous or not-S → SS; S → a; S → b

Solution-

Let us consider a string w generated by the given grammar-w = abba

Now, let us draw parse trees for this string w.



Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Problem 6:

Check whether the given grammar is ambiguous or not-

S → A / B

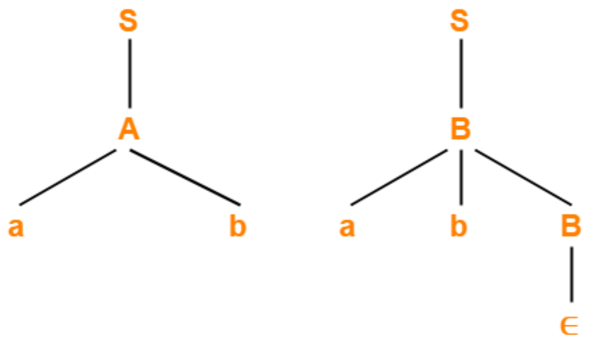
A → aAb / ab

B → abB / ε

Solution:

Let us consider a string w generated by the given grammar-w = ab

Now, let us draw parse trees for this string w.



Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Problem7:

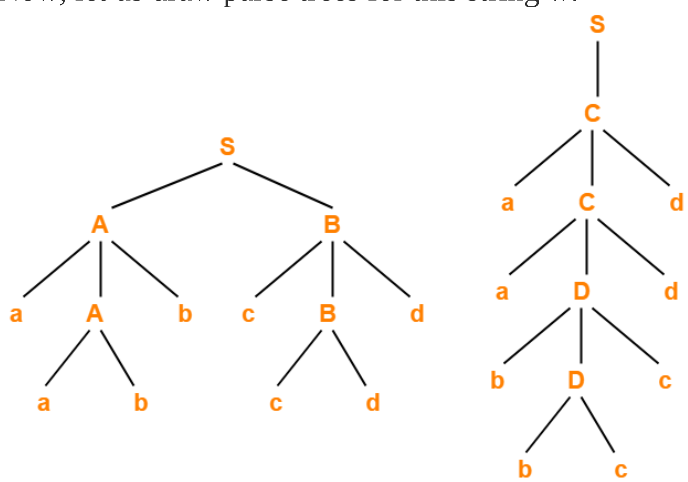
Check whether the given grammar is ambiguous or not-

$$S \rightarrow AB \,/\, C$$
$$A \rightarrow aAb \,/\, ab$$
$$B \rightarrow cBd \,/\, cd$$
$$C \rightarrow aCd \,/\, aDd$$
$$D \rightarrow bDc \,/\, bc$$

Solution-

Let us consider a string w generated by the given grammar-w = aabbccdd

Now, let us draw parse trees for this string w.



Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Problem 8:

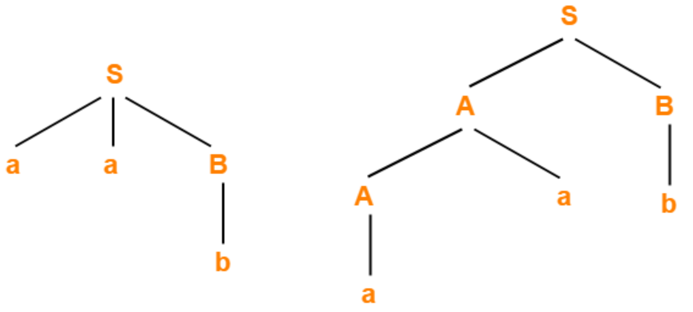
Check whether the given grammar is ambiguous or not-

$$S \rightarrow AB \,/\, aaB$$
$$A \rightarrow a \,/\, Aa$$
$$B \rightarrow b$$

Solution:

Let us consider a string w generated by the given grammar-w = aab

Now, let us draw parse trees for this string w.



Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Problem-9:

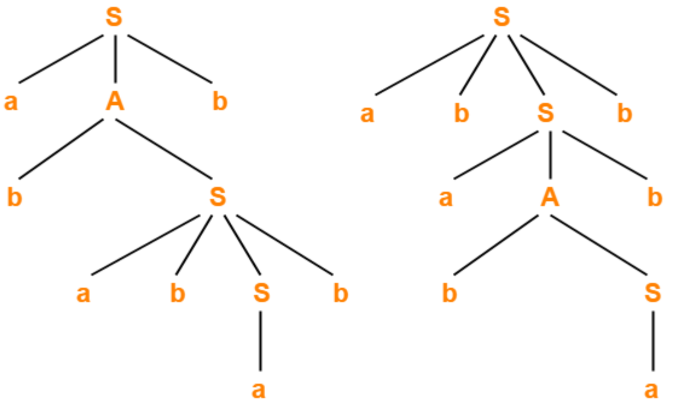
Check whether the given grammar is ambiguous or not-

$$S \rightarrow a \,/\, abSb \,/\, aAb$$
$$A \rightarrow bS \,/\, aAAb$$

Solution:

Let us consider a string w generated by the given grammar-w = abababb

Now, let us draw parse trees for this string w.



Parse tree-01 Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Problem 10:

Check whether the given grammar G is ambiguous or not.

1. E → E + E
2. E → E - E
3. E → id

Solution:

From the above grammar String "id + id - id" can be derived in 2 ways:

First Leftmost derivation

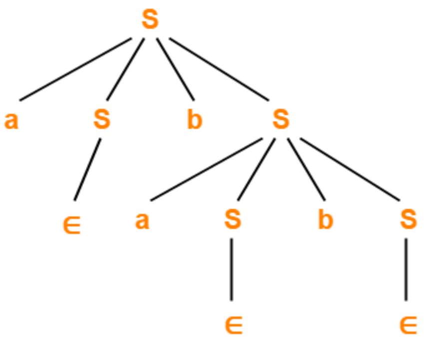
1. E ⇒ E + E
2. ⇒ id + E
3. ⇒ id + E - E
4. ⇒ id + id - E
5. ⇒ id + id- id

Second Leftmost derivation

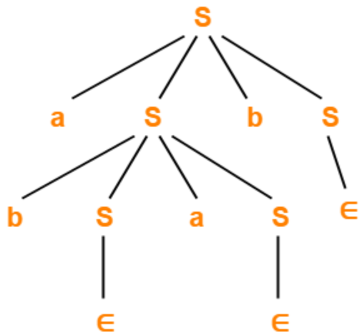
1. E ⇒ E - E
2. ⇒ E + E - E
3. ⇒ id + E - E
4. ⇒ id + id - E
5. ⇒ id + id - id

Since there are two leftmost derivation for a single string "id + id - id", the grammar G is ambiguous.

Problem11:
Check whether the given grammar is ambiguous or not-S → aSbS / bSaS / ε
Solution:
Let us consider a string w generated by the given grammar-w = abab
Now, let us draw parse trees for this string w.



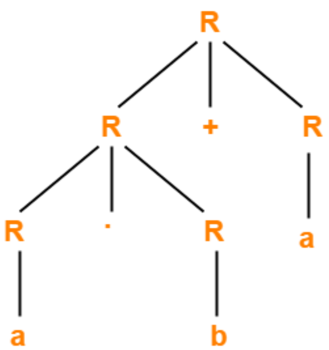
Parse tree-01



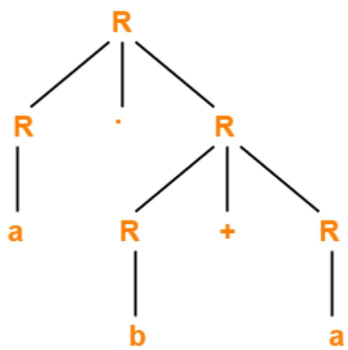
Parse tree-02

Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Problem 12:
Check whether the given grammar is ambiguous or not-R → R + R / R . R / R* / a / b
Solution:
Let us consider a string w generated by the given grammar-w = ab + a
Now, let us draw parse trees for this string w.



Parse tree-01



Parse tree-02

For Practice:

1. Check whether the grammar G with production rules −X → X+X | X*X |X| a is ambiguous or not.
2. Check whether the given grammar G is ambiguous or not.
   S → aSb | SS
   S → ε
3. Check that the given grammar is ambiguous or not. Also, find an equivalent unambiguous grammar.
   1. S → S + S
   2. S → S * S
   3. S → S ^ S
   4. S → a
4. Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.
   - S → ABA
   - A → aA | ε
   - B → bB | ε

1.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF SCIENCE AND HUMANITIES

DEPARTMENT OF MATHEMATICS

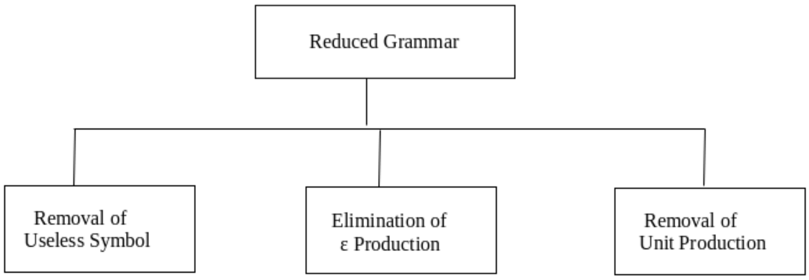
UNIT – II - Formal Language and Automata – SMT1404

Introduction

Simplification of CFG

As we have seen, various languages can efficiently be represented by a context-free grammar. All the grammar are not always optimized that means the grammar may consist of some extra symbols(non-terminal). Extra symbols, unnecessarily 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.
4. Let us study the reduction process in detail.



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
2. S → YX
   Similarly if the last X in R.H.S. = ε. Then
3. S → XY
   If Y = ε then
4. S → XX
   If Y and X are ε then,
5. S → X
   If both X are replaced by ε
6. S → Y
   Now,
7. S → XY | YX | XX | X | Y
   Now let us consider
   X → 0X
   If we place ε at right-hand side for X then,
8. X → 0
9. X → 0X | 0
10. Similarly Y → 1Y | 1
    Collectively we can rewrite the CFG with removed ε production as
11. S → XY | YX | XX | X | Y
12. X → 0X | 0
13. 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. $B \to 1 \mid A$
4. $C \to 01$

Solution:

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

1. $S \to 0A \mid 1B \mid 01$

Similarly, $B \to A$ is also a unit production so we can modify it as

1. $B \to 1 \mid 0S \mid 00$
2. Thus finally we can write CFG without unit production as
3. $S \to 0A \mid 1B \mid 01$
4. $A \to 0S \mid 00$
5. $B \to 1 \mid 0S \mid 00$
6. $C \to 01$

Chomsky Normal Forms:

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, $A \to \varepsilon$.
- A non-terminal generating two non-terminals. For example, $S \to AB$.
- A non-terminal generating a terminal. For example, $S \to a$.

For example:

1. $G_1 = \{S \to AB, S \to c, A \to a, B \to b\}$
2. $G_2 = \{S \to aA, A \to a, B \to c\}$

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 $S \to aZ$ 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. $S_1 \to S$, where S_1 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 $S \to aA$ can be decomposed as:

1. $S \to RA$
2. $R \to a$

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

1. $S \to RS$
2. $R \to AS$

Problem1:

Convert the given CFG to CNF. Consider the given grammar G_1:

1. $S \to a \mid aA \mid B$
2. $A \to aBB \mid \varepsilon$

3. $B \rightarrow Aa \mid b$

Solution:

Step 1: We will create a new production $S_1 \rightarrow S$, as the start symbol S appears on the RHS. The grammar will be:

1. $S_1 \rightarrow S$
2. $S \rightarrow a \mid aA \mid B$
3. $A \rightarrow aBB \mid \varepsilon$
4. $B \rightarrow Aa \mid b$

Step 2: As grammar G_1 contains $A \rightarrow \varepsilon$ null production, its removal from the grammar yields:

1. $S1 \rightarrow S$
2. $S \rightarrow a \mid aA \mid B$
3. $A \rightarrow aBB$
4. $B \rightarrow Aa \mid b \mid a$

Now, as grammar G_1 contains Unit production $S \rightarrow B$, its removal yields:

1. $S_1 \rightarrow S$
2. $S \rightarrow a \mid aA \mid Aa \mid b$
3. $A \rightarrow aBB$
4. $B \rightarrow Aa \mid b \mid a$

Also remove the unit production $S_1 \rightarrow S$, its removal from the grammar yields:

1. $S_0 \rightarrow a \mid aA \mid Aa \mid b$
2. $S \rightarrow a \mid aA \mid Aa \mid b$
3. $A \rightarrow aBB$
4. $B \rightarrow Aa \mid b \mid a$

Step 3: In the production rule $S_0 \rightarrow aA \mid Aa$, $S \rightarrow aA \mid Aa$, $A \rightarrow aBB$ and $B \rightarrow Aa$, terminal a exists on RHS with non-terminals. So we will replace terminal a with X:

1. $S_0 \rightarrow a \mid XA \mid AX \mid b$
2. $S \rightarrow a \mid XA \mid AX \mid b$
3. $A \rightarrow XBB$
4. $B \rightarrow AX \mid b \mid a$
5. $X \rightarrow a$

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

1. $S0 \rightarrow a \mid XA \mid AX \mid b$
2. $S \rightarrow a \mid XA \mid AX \mid b$
3. $A \rightarrow RB$
4. $B \rightarrow AX \mid b \mid a$
5. $X \rightarrow a$
6. $R \rightarrow XB$

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

Problem 2:

Convert the following CFG into CNF $S \rightarrow ASA \mid aB$, $A \rightarrow B \mid S$, $B \rightarrow b \mid \varepsilon$

Solution:

(1) Since **S** appears in R.H.S, we add a new state $\mathbf{S_0}$ and $\mathbf{S_0 \rightarrow S}$ is added to the production set and it becomes

$S_0 \rightarrow S$, $S \rightarrow ASA \mid aB$, $A \rightarrow B \mid S$, $B \rightarrow b \mid \varepsilon$

(2) Now we will remove the null productions:

$B \rightarrow \varepsilon$ and $A \rightarrow \varepsilon$

After removing $B \rightarrow \varepsilon$, the production set becomes

$S_0 \rightarrow S$, $S \rightarrow ASA \mid aB \mid a$, $A \rightarrow B \mid S \mid \in$, $B \rightarrow b$

After removing $A \rightarrow \in$, the production set becomes

$S_0 \rightarrow S$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA \mid S$, $A \rightarrow B \mid S$, $B \rightarrow b$

(3) Now we will remove the unit productions.

After removing $S \rightarrow S$, the production set becomes

$S_0 \rightarrow S$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $A \rightarrow B \mid S$, $B \rightarrow b$

After removing $S_0 \rightarrow S$, the production set becomes

$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$

$A \rightarrow B \mid S$, $B \rightarrow b$

After removing $A \rightarrow B$, the production set becomes −

$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$

$A \rightarrow S \mid b$

$B \rightarrow b$

After removing $A \rightarrow S$, the production set becomes

$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$

$A \rightarrow b \mid ASA \mid aB \mid a \mid AS \mid SA$, $B \rightarrow b$

(4) Now we will find out more than two variables in the R.H.S

Here, $S_0 \rightarrow ASA$, $S \rightarrow ASA$, $A \rightarrow ASA$ violates two Non-terminals in R.H.S.

Hence we will apply step 4 and step 5 to get the following final production set which is in CNF.

$S_0 \rightarrow AX \mid aB \mid a \mid AS \mid SA$

$S \rightarrow AX \mid aB \mid a \mid AS \mid SA$

$A \rightarrow b \mid AX \mid aB \mid a \mid AS \mid SA$

$B \rightarrow b$

$X \rightarrow SA$

(5) We have to change the productions $S_0 \rightarrow aB$, $S \rightarrow aB$, $A \rightarrow aB$

And the final production set becomes

$S_0 \rightarrow AX \mid YB \mid a \mid AS \mid SA$

$S \rightarrow AX \mid YB \mid a \mid AS \mid SA$

$A \rightarrow b$ $A \rightarrow b \mid AX \mid YB \mid a \mid AS \mid SA$

$B \rightarrow b$

$X \rightarrow SA$

$Y \rightarrow a$

Problem 3:
Convert the given grammar to CNF $S \to aAD$; $A \to aB / bAB$; $B \to b$; $D \to d$
Solution:
Step-01:
The given grammar is already completely reduced.

Step-02:
The productions already in Chomsky normal form are-
$B \to b$ ………..(1)
$D \to d$ ………..(2)
These productions will remain as they are.

The productions not in Chomsky normal form are
$S \to aAD$ ………..(3)
$A \to aB / bAB$ ………..(4)
We will convert these productions in Chomsky normal form.

Step-03:
Replace the terminal symbols a and b by new variables C_a and C_b.
This is done by introducing the following two new productions in the grammar-
$C_a \to a$ ………..(5)
$C_b \to b$ ………..(6)
Now, the productions (3) and (4) modifies to-
$S \to C_aAD$ ………..(7)
$A \to C_aB / C_bAB$ ………..(8)

Step-04:
Replace AD and AB by new variables C_{AD} and C_{AB} respectively.
This is done by introducing the following two new productions in the grammar
$C_{AD} \to AD$ ………..(9)
$C_{AB} \to AB$ ………..(10)
Now, the productions (7) and (8) modifies to-
$S \to C_aC_{AD}$ ………..(11)
$A \to C_aB / C_bC_{AB}$ ………..(12)

Step-05:
From (1), (2), (5), (6), (9), (10), (11) and (12), the resultant grammar is-
$S \to C_aC_{AD}$
$A \to C_aB / C_bC_{AB}$
$B \to b$
$D \to d$
$C_a \to a$
$C_b \to b$
$C_{AD} \to AD$
$C_{AB} \to AB$
This grammar is in Chomsky normal form.

Problem 4:

Convert the given grammar to CNF-S → 1A / 0B; A → 1AA / 0S / 0; B → 0BB / 1S / 1

Solution:

Step1:

The given grammar is already completely reduced.

Step2:

The productions already in chomsky normal form are-

A → 0 ………..(1)

B → 1 ………..(2)

These productions will remain as they are.

The productions not in chomsky normal form are-

S → 1A / 0B ………..(3)

A → 1AA / 0S ………..(4)

B → 0BB / 1S ………..(5)

We will convert these productions in Chomsky normal form.

Step3:

Replace the terminal symbols 0 and 1 by new variables C and D.

This is done by introducing the following two new productions in the grammar-

C → 0 ………..(6)

D → 1 ………..(7)

Now, the productions (3), (4) and (5) modifies to-

S → DA / CB ………..(8)

A → DAA / CS ………..(9)

B → CBB / DS ………..(10)

Step4:

Out of (8), (9) and (10), the productions already in Chomsky Normal Form are-

S → DA / CB ………..(11)

A → CS ………..(12)

B → DS ………..(13)

These productions will remain as they are.

The productions not in Chomsky normal form are-

A → DAA ………..(14)

B → CBB ………..(15)

We will convert these productions in Chomsky Normal Form.

Step 5:

Replace AA and BB by new variables E and F respectively.

This is done by introducing the following two new productions in the grammar-

E → AA ………..(16)

F → BB ………..(17)

Now, the productions (14) and (15) modifies to-

A → DE ………..(18)

B → CF ………..(19)

Step 6:

From (1), (2), (6), (7), (11), (12), (13), (16), (17), (18) and (19), the resultant grammar is-

S → DA / CB

A → CS / DE / 0
B → DS / CF / 1
C → 0
D → 1
E → AA
F → BB
This grammar is in Chomsky normal form.

Pumping Lemma:

If **L** is a context-free language, there is a pumping length **p** such that any string **w ∈ L** of length ≥ **p** can be written as **w = uvxyz**, where **vy ≠ ε**, **|vxy| ≤ p**, and for all **i ≥ 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: 5

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| ≤ n and vx ≠ ε.
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.

References:

1. John E Hopcroft, Jeffery D, Ullman, Introduction to Automata Theory and Computations, Narosa Publishing House.
2. Dr. M. Venkatraman, Dr. N. Sridharan, N. Chandrasekaran," Discrete Mathematics", Chapter 12, Sections 1 to 12.
3. https://www.cs.odu.edu/~toida/nerzic/390teched/regular/fa/nfa-definitions.html
4. https://www.tutorialspoint.com/automata_theory/ndfa_to_dfa_conversion.htm
5. https://en.wikipedia.org/wiki/Deterministic_finite_automaton