1.1 INTRODUCTION

Fundamentals

• Symbol –

An atomic unit, such as a digit, character, lower-case letter, etc. Sometimes a word. [Formal language does not deal with the "meaning" of the symbols.]

• Alphabet –

A finite set of symbols, usually denoted by Σ.

Σ = {0, 1} Σ = {0, a, 9, 4} Σ = {a, b, c, d}

• String –

A finite length sequence of symbols, presumably from some alphabet.

w = 0110 y = 0aa x = aabcaa z = 111

Special string:

ε (also denoted by λ)

Concatenation: wz = 0110111

Length: |w| = 4 |ε| = 0 |x| = 6

Reversal: y R = aa0

• Some special sets of strings:

Σ * All strings of symbols from Σ Σ + Σ * - {ε}

• Example: Σ = {0, 1} Σ * = {ε, 0, 1, 00, 01, 10, 11, 000, 001,…}

Σ + = {0, 1, 00, 01, 10, 11, 000, 001,…}

• A language is:

1) A set of strings from some alphabet (finite or infinite). In other words,

2) Any subset L of Σ*

• Some special languages:

{} The empty set/language, containing no string.

{ε} A language containing one string, the empty string.

• Examples: Σ = {0, 1} L = {x | x is in Σ* and x contains an even number of 0's}

Σ = {0, 1, 2,…, 9, .} L = {x | x is in Σ* and x forms a finite length real number} = {0, 1.5, 9.326,…}

Σ = {a, b, c,…, z, A, B,…, Z} L = {x | x is in Σ* and x is a Pascal reserved word}
32 = {BEGIN, END, IF,…} Σ = {Pascal reserved words} U { (,), ., :, ;,…} U {Legal Pascal identifiers} L = {x | x is in Σ* and x is a syntactically correct Pascal program}

‘

Σ = {English words} L = {x | x is in Σ^* and x is a syntactically correct English sentence}

‘

Automata Tutorial

Theory of automata is a theoretical branch of computer science and mathematical. It is the study of abstract machines and the computation problems that can be solved using these machines. The abstract machine is called the automata. An automaton with a finite number of states is called a Finite automaton.

Theory of Automata

Theory of automata is a theoretical branch of computer science and mathematical. It is the study of abstract machines and the computation problems that can be solved using these machines. The abstract machine is called the automata. The main motivation behind developing the automata theory was to develop methods to describe and analyse the dynamic behaviour of discrete systems.

This automaton consists of states and transitions. The **State** is represented by **circles**, and the **Transitions** is represented by **arrows**.

Automata is the kind of machine which takes some string as input and this input goes through a finite number of states and may enter in the final state.

There are the basic terminologies that are important and frequently used in automata:

Symbols:

Symbols are an entity or individual objects, which can be any letter, alphabet or any picture.

Example:

‘

1, a, b, #

Alphabets:

Alphabets are a finite set of symbols. It is denoted by ∑.

Examples:

1. ∑ = {a, b}
2.
3. ∑ = {A, B, C, D}
4.
5. ∑ = {0, 1, 2}
6.
7. ∑ = {0, 1, ..., 5]
8.
9. ∑ = {#, β, Δ}

String:

It is a finite collection of symbols from the alphabet. The string is denoted by w.

Example 1:

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

- A string with zero occurrences of symbols is known as an empty string. It is represented by ε.
- The number of symbols in a string w is called the length of a string. It is denoted by |w|.

Example 2:

1. w = 010
2.
3. Number of Sting |w| = 3

Lang A set of strings all of which are chosen form some Σ^*, where Σ is a particular alphabet, is called a language. If Σ is an alphabet and $L \subseteq \Sigma^*$, then L named as language over alphabet Σ. Informally a language is an equivalent

‘

member of the power set of Σ^* or any subsets of the Kleene closure of an alphabet Σ can be considered as languages.uage:

A language is a collection of appropriate string. A language which is formed over Σ can be **Finite** or **Infinite**.

Example: 1

```
L1 = {Set of string of length 2}

     = {aa, bb, ba, bb}
```

Finite Language

Example: 2

```
L2 = {Set of all strings starts with 'a'}

     = {a, aa, aaa, abb, abbb, ababb}
```

Infinite Language

- **Languages: "A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols"**
-

Grammars: "A grammar can be regarded as a device that enumerates the sentences of a language" - nothing more, nothing less.

Grammar:

A grammar is a set of rules for a strings generation in a formal language. These rules describe how does strings forms from the language that are valid according to the language syntax. A grammar does not describe the meaning of the generated string.

For example:

S → SbS
a → a|ε
b → b|ε

‘

Chomsky Hierarchy

Chomsky Hierarchy represents the class of languages that are accepted by the different machine. The category of language in Chomsky's Hierarchy is as given below:

1. Type 0 known as Unrestricted Grammar.
2. Type 1 known as Context Sensitive Grammar.
3. Type 2 known as Context Free Grammar.
4. Type 3 Regular Grammar.

This is a hierarchy. Therefore every language of type 3 is also of type 2, 1 and 0. Similarly, every language of type 2 is also of type 1 and type 0, etc.

Type 0 Grammar:

Type 0 grammar is known as Unrestricted grammar. There is no restriction on the grammar rules of these types of languages. These languages can be efficiently modeled by Turing machines.

For example:

1. bAa → aa
2. S → s

Type 1 Grammar:

Type 1 grammar is known as Context Sensitive Grammar. The context sensitive grammar is used to represent context sensitive language. The context sensitive grammar follows the following rules:

- The context sensitive grammar may have more than one symbol on the left hand side of their production rules.
- The number of symbols on the left-hand side must not exceed the number of symbols on the right-hand side.

‘

- The rule of the form A → ε is not allowed unless A is a start symbol. It does not occur on the right-hand side of any rule.
- The Type 1 grammar should be Type 0. In type 1, Production is in the form of V → T

Where the count of symbol in V is less than or equal to T.

For example:

1. S → AT
2. T → xy
3. A → a

Type 2 Grammar:

Type 2 Grammar is known as Context Free Grammar. Context free languages are the languages which can be represented by the context free grammar (CFG). Type 2 should be type 1. The production rule is of the form

1. A → α

Where A is any single non-terminal and is any combination of terminals and non-terminals.

For example:

1. A → aBb
2. A → b
3. B → a

Type 3 Grammar:

Type 3 Grammar is known as Regular Grammar. Regular languages are those languages which can be described using regular expressions. These languages can be modeled by NFA or DFA.

Type 3 is most restricted form of grammar. The Type 3 grammar should be Type 2 and Type 1. Type 3 should be in the form of

1. V → T*V / T*

For example:

‘

1. A → xy

Finite Automata

- Finite automata are used to recognize patterns.
- It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
- At the time of transition, the automata can either move to the next state or stay in the same state.
- 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.

Formal Definition of FA

A finite automaton is a collection of 5-tuple (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

Finite Automata Model:

Finite automata can be represented by input tape and finite control.

Input tape: It is a linear tape having some number of cells. Each input symbol is placed in each cell.

Finite control: The finite control decides the next state on receiving particular input from input tape. The tape reader reads the cells one by one from left to right, and at a time only one input symbol is read.

‘

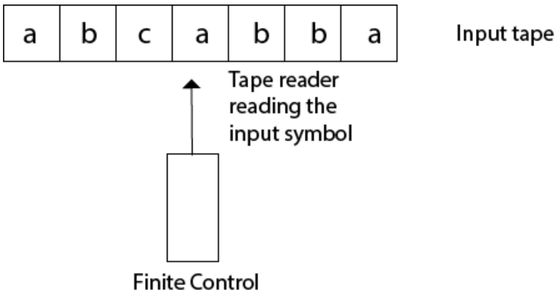
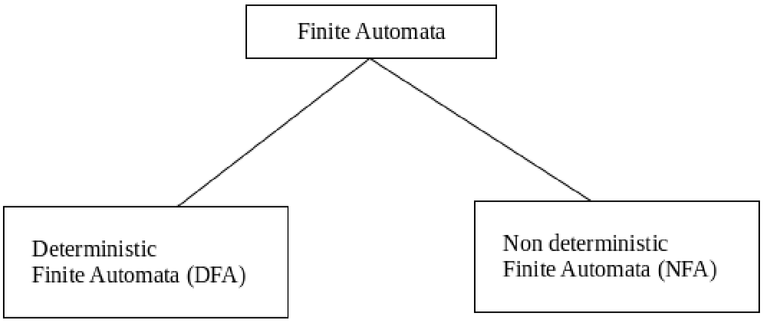


Fig :- Finite automata model

Types of Automata:

There are two types of finite automata:

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



1. DFA

DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. In the DFA, the machine goes to one state only for a particular input character. DFA does not accept the null move.

2. NFA

NFA stands for non-deterministic finite automata. It is used to transmit any number of states for a particular input. It can accept the null move.

Some important points about DFA and NFA:

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.

Transition Diagram

A transition diagram or state transition diagram is a directed graph which can be constructed as follows:

- There is a node for each state in Q, which is represented by the circle.
- There is a directed edge from node q to node p labeled a if δ(q, a) = p.
- In the start state, there is an arrow with no source.
- Accepting states or final states are indicating by a double circle.

Some Notations that are used in the transition diagram:

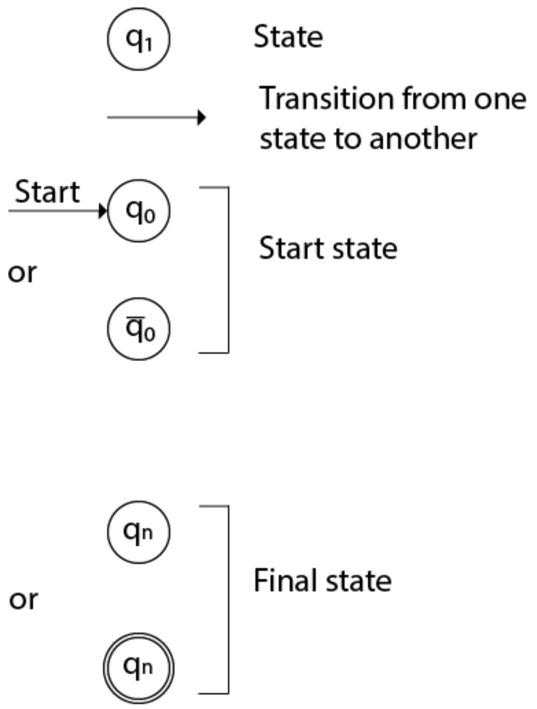


Fig:- Notations

There is a description of how a DFA operates:

1. In DFA, the input to the automata can be any string. Now, put a pointer to the start state q and read the input string w from left to right and move the pointer according to the transition function, δ. We can read one symbol at a time. If the next symbol of

‘

string w is a and the pointer is on state p, move the pointer to δ(p, a). When the end of the input string w is encountered, then the pointer is on some state F.

2. The string w is said to be accepted by the DFA if r ∈ F that means the input string w is processed successfully and the automata reached its final state. The string is said to be rejected by DFA if r ∉ F.

Example 1:

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

Solution:

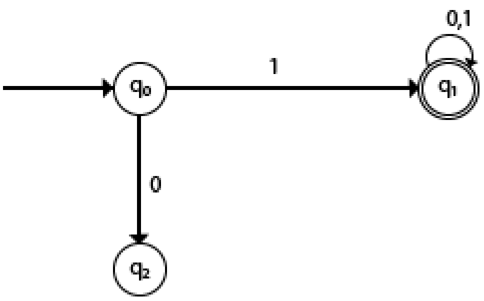


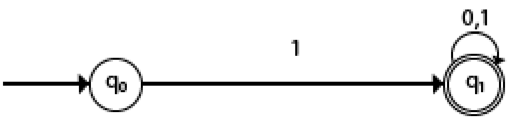
Fig: Transition diagram

The finite automata can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q0 on receiving 0, the machine changes its state to q2, which is the dead state. From q1 on receiving input 0, 1 the machine changes its state to q1, 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.

Example 2:

NFA with ∑ = {0, 1} accepts all strings starting with 1.

Solution:



The NFA can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q1 on receiving input 0, 1 the machine changes its state to q1. The possible input string that can be generated is 10, 11, 110, 101, 111......, that means all string starts with 1.

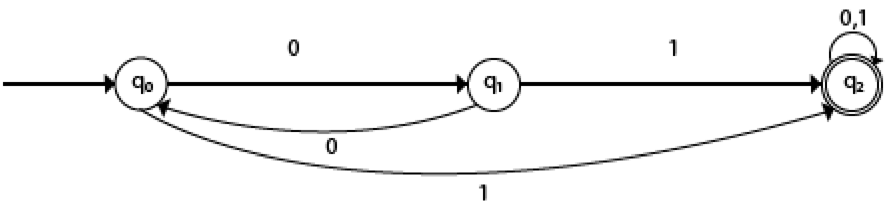
Transition Table

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:

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

Example 1:



Solution:

Transition table of given DFA is as follows:

Present State	Next state for Input 0	Next State of Input 1
→q0	q1	q2
q1	q0	q2
*q2	q2	q2

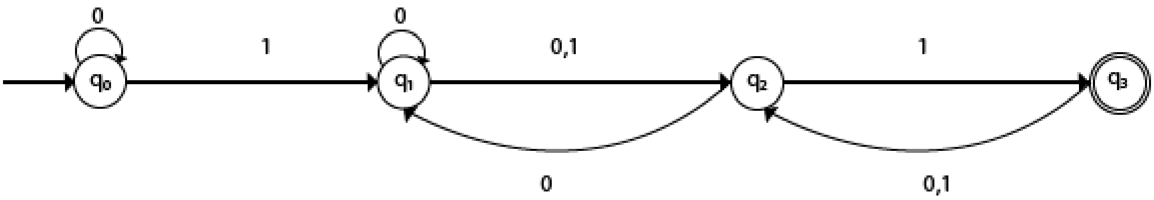
Explanation:

- In the above table, the first column indicates all the current states. Under column 0 and 1, the next states are shown.

‘

- The first row of the transition table can be read as, when the current state is q0, on input 0 the next state will be q1 and on input 1 the next state will be q2.
- In the second row, when the current state is q1, on input 0, the next state will be q0, and on 1 input the next state will be q2.
- In the third row, when the current state is q2 on input 0, the next state will be q2, and on 1 input the next state will be q2.
- The arrow marked to q0 indicates that it is a start state and circle marked to q2 indicates that it is a final state.

Example 2:



Solution:

Transition table of given NFA is as follows:

Present State	Next state for Input 0	Next State of Input 1
→q0	q0	q1
q1	q1, q2	q2
q2	q1	q3
*q3	q2	q2

Explanation:

- The first row of the transition table can be read as, when the current state is q0, on input 0 the next state will be q0 and on input 1 the next state will be q1.
- In the second row, when the current state is q1, on input 0 the next state will be either q1 or q2, and on 1 input the next state will be q2.

‘

- In the third row, when the current state is q2 on input 0, the next state will be q1, and on 1 input the next state will be q3.
- In the fourth row, when the current state is q3 on input 0, the next state will be q2, and on 1 input the next state will be q2.

DFA (Deterministic finite automata)

- DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time.
- In DFA, there is only one path for specific input from the current state to the next state.
- DFA does not accept the null move, i.e., the DFA cannot change state without any input character.
- DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

In the following diagram, we can see that from state q0 for input a, there is only one path which is going to q1. Similarly, from q0, there is only one path for input b going to q2.

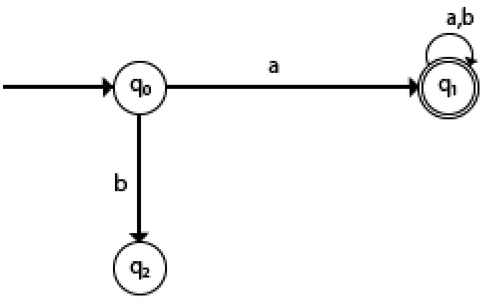


Fig:- DFA

Formal Definition of DFA

A DFA is a collection of 5-tuples same as we described in the definition of FA.

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

‘

Transition function can be defined as:

1. δ: Q x ∑→Q

Graphical Representation of DFA

A DFA can be represented by digraphs called state diagram. In which:

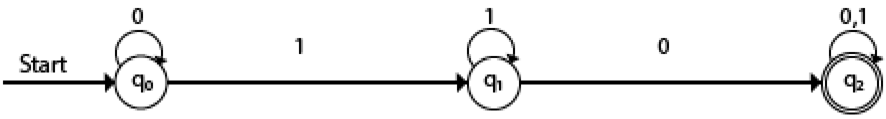
1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by a double circle.

Example 1:

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 = {q0}
4. F = {q2}

Solution:

Transition Diagram:



Transition Table:

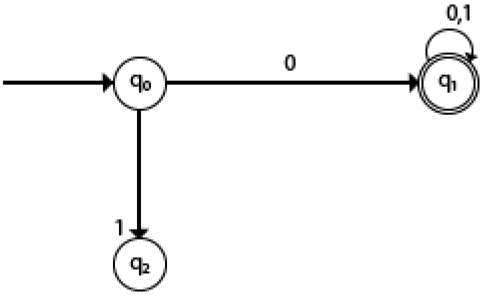
Present State	Next state for Input 0	Next State of Input 1
→q0	q0	q1
q1	q2	q1
*q2	q2	q2

Example 2:

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

‘

Solution:



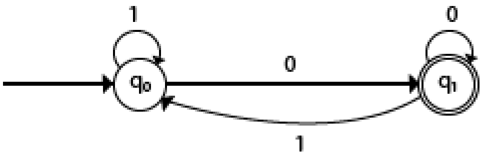
Explanation:

- In the above diagram, we can see that on given 0 as input to DFA in state q0 the DFA changes state to q1 and always go to final state q1 on starting input 0. It can accept 00, 01, 000, 001....etc. It can't accept any string which starts with 1, because it will never go to final state on a string starting with 1.

Example 3:

DFA with ∑ = {0, 1} accepts all ending with 0.

Solution:



Explanation:

In the above diagram, we can see that on given 0 as input to DFA in state q0, the DFA changes state to q1. It can accept any string which ends with 0 like 00, 10, 110, 100....etc. It can't accept any string which ends with 1, because it will never go to the final state q1 on 1 input, so the string ending with 1, will not be accepted or will be rejected.

Examples of DFA

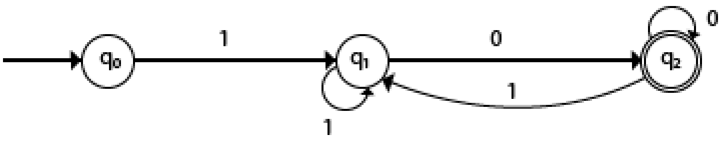
Example 1:

Design a FA with ∑ = {0, 1} accepts those string which starts with 1 and ends with 0.

‘

Solution:

The FA will have a start state q0 from which only the edge with input 1 will go to the next state.



In state q1, if we read 1, we will be in state q1, but if we read 0 at state q1, we will reach to state q2 which is the final state. In state q2, if we read either 0 or 1, we will go to q2 state or q1 state respectively. Note that if the input ends with 0, it will be in the final state.

Example 2:

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

Solution:

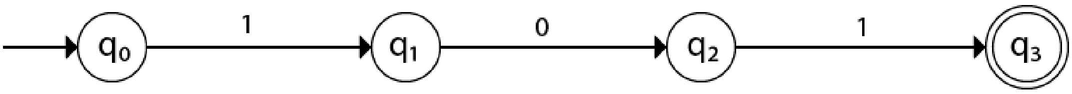


Fig: FA

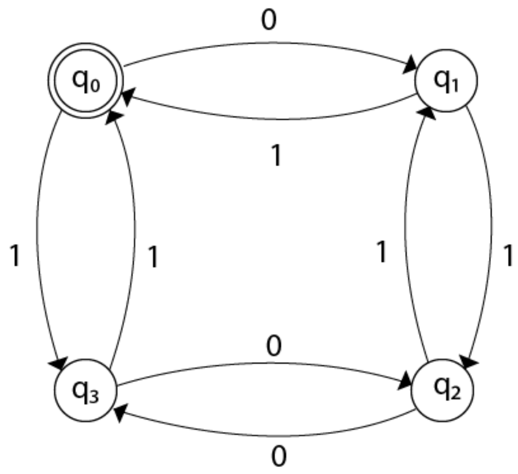
In the given solution, we can see that only input 101 will be accepted. Hence, for input 101, there is no other path shown for other input.

Example 3:

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 q0 is a start state and the final state also. Note carefully that a symmetry of 0's and 1's is maintained. We can associate meanings to each state as:

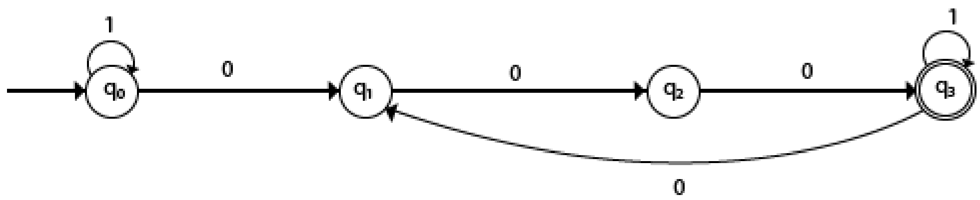
q0: state of even number of 0's and even number of 1's.
q1: state of odd number of 0's and even number of 1's.
q2: state of odd number of 0's and odd number of 1's.
q3: state of even number of 0's and odd number of 1's.

Example 4:

Design FA with ∑ = {0, 1} accepts the set of all strings with three consecutive 0's.

Solution:

The strings that will be generated for this particular languages are 000, 0001, 1000, 10001, ... in which 0 always appears in a clump of 3. The transition graph is as follows:



Note that the sequence of triple zeros is maintained to reach the final state.

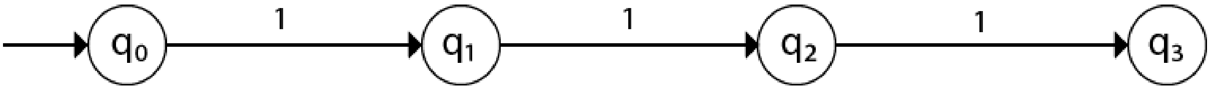
Example 5:

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

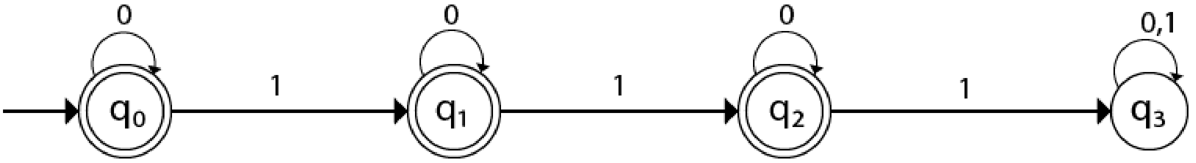
‘

Solution:

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



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



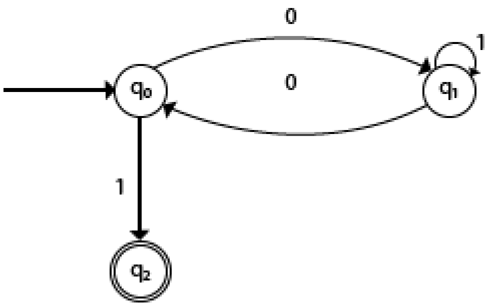
The stages q0, q1, q2 are the final states. The DFA will generate the strings that do not contain consecutive 1's like 10, 110, 101,..... etc.

Example 6:

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:



NFA (Non-Deterministic finite automata)

- NFA stands for non-deterministic finite automata. It is easy to construct an NFA than DFA for a given regular language.

- The finite automata are called NFA when there exist many paths for specific input from the current state to the next state.
- Every NFA is not DFA, but each NFA can be translated into DFA.
- 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 q0 for input a, there are two next states q1 and q2, similarly, from q0 for input b, the next states are q0 and q1. 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.

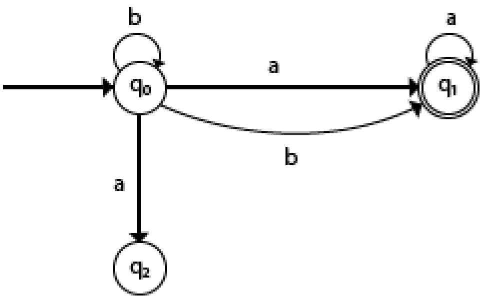


Fig:- NDFA

Formal definition of NFA:

NFA also has five states same as DFA, but with different transition function, as shown follows:

$\delta: Q \times \sum \rightarrow 2^Q$

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

Graphical Representation of an NFA

An NFA can be represented by digraphs called state diagram. In which:

1. The state is represented by vertices.

‘

2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by the double circle.

Example 1:

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 = {q0}
4. F = {q2}

Solution:

Transition diagram:

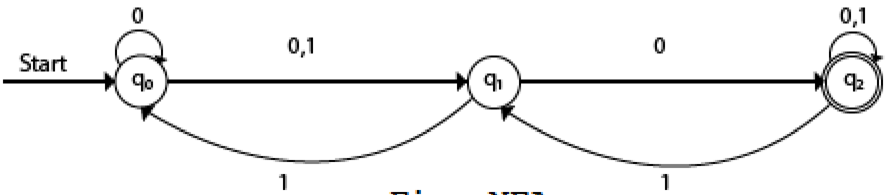


Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→q0	q0, q1	q1
q1	q2	q0
*q2	q2	q1, q2

In the above diagram, we can see that when the current state is q0, on input 0, the next state will be q0 or q1, and on 1 input the next state will be q1. When the current state is q1, on input 0 the next state will be q2 and on 1 input, the next state will be q0. When the current state is q2, on 0 input the next state is q2, and on 1 input the next state will be q1 or q2.

Example 2:

NFA with ∑ = {0, 1} accepts all strings with 01.

‘

Solution:

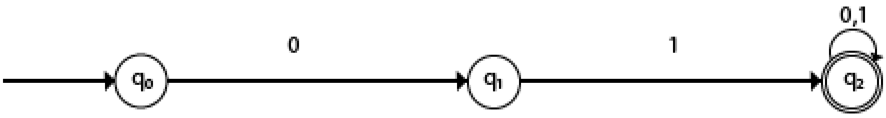


Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→q0	q1	ε
q1	ε	q2
*q2	q2	q2

Example 3:

NFA with ∑ = {0, 1} and accept all string of length atleast 2.

Solution:

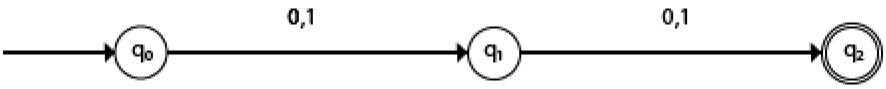


Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→q0	q1	q1
q1	q2	q2
*q2	ε	ε

Examples of NFA

‘

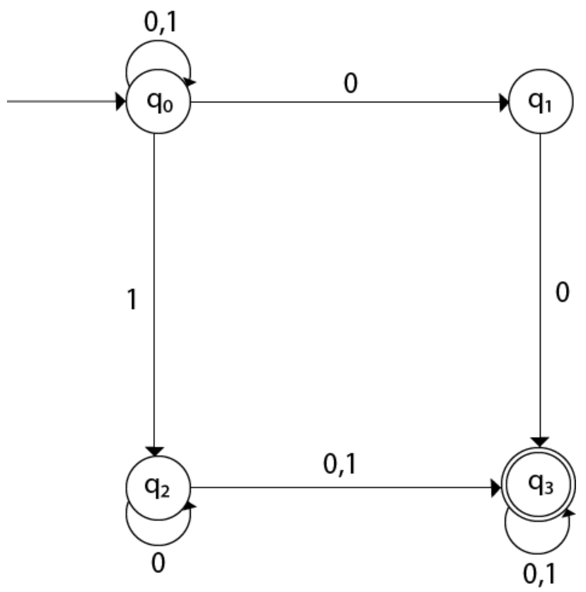
Example 1:

Design a NFA for the transition table as given below:

Present State	0	1
→q0	q0, q1	q0, q2
q1	q3	ε
q2	q2, q3	q3
→q3	q3	q3

Solution:

The transition diagram can be drawn by using the mapping function as given in the table.



Here,

1. δ(q0, 0) = {q0, q1}
2. δ(q0, 1) = {q0, q2}
3. Then, δ(q1, 0) = {q3}
4. Then, δ(q2, 0) = {q2, q3}
5. δ(q2, 1) = {q3}
6. Then, δ(q3, 0) = {q3}
7. δ(q3, 1) = {q3}

‘

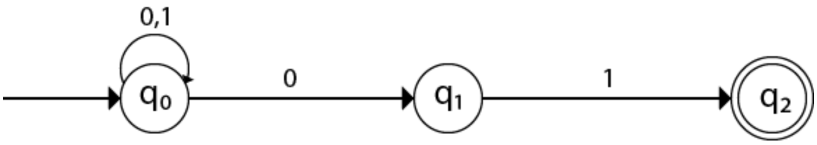
Example 2:

Design an NFA with ∑ = {0, 1} accepts all string ending with 01.

Solution:

Anything either 0 or 1	0 1

Hence, NFA would be:

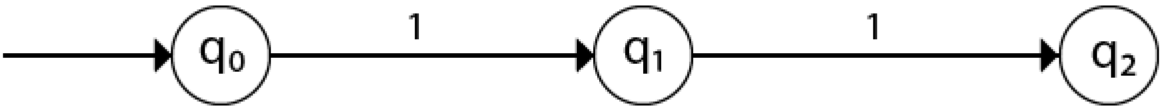


Example 3:

Design an NFA with ∑ = {0, 1} in which double '1' is followed by double '0'.

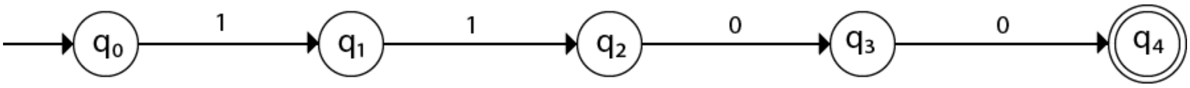
Solution:

The FA with double 1 is as follows:



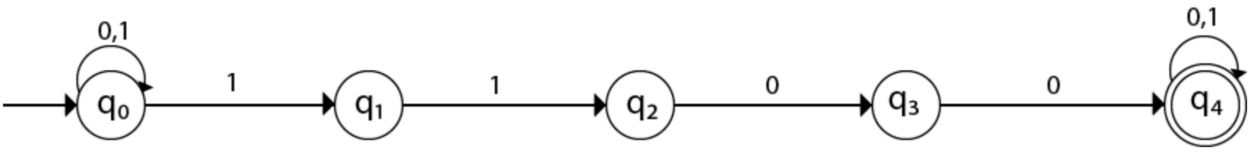
It should be immediately followed by double 0.

Then,



Now before double 1, there can be any string of 0 and 1. Similarly, after double 0, there can be any string of 0 and 1.

Hence the NFA becomes:



Now considering the string 01100011

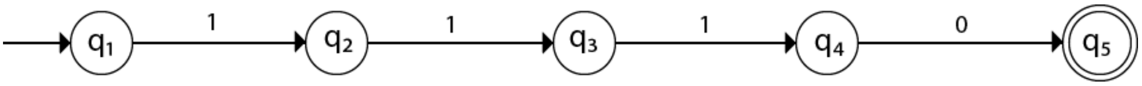
1. q0 → q1 → q2 → q3 → q4 → q4 → q4 → q4

Example 4:

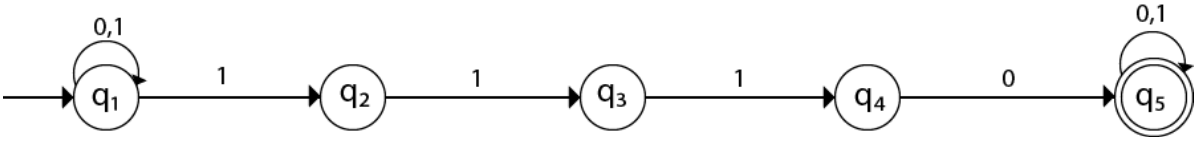
Design an NFA in which all the string contain a substring 1110.

Solution:

The language consists of all the string containing substring 1010. The partial transition diagram can be:



Now as 1010 could be the substring. Hence we will add the inputs 0's and 1's so that the substring 1010 of the language can be maintained. Hence the NFA becomes:



Transition table for the above transition diagram can be given below:

Present State	0	1
→q1	q1	q1, q2
q2		q3
q3		q4
q4	q5	
*q5	q5	q5

‘

Consider a string 111010,

1. δ(q1, 111010) = δ(q1, 1100)
2. = δ(q1, 100)
3. = δ(q2, 00)

Got stuck! As there is no path from q2 for input symbol 0. We can process string 111010 in another way.

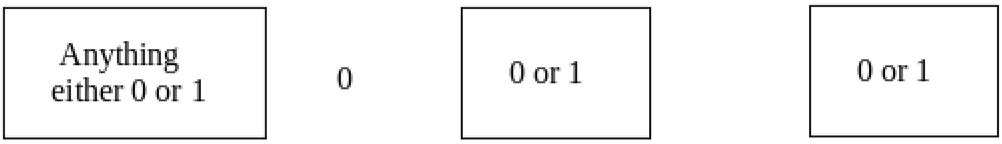
1. δ(q1, 111010) = δ(q2, 1100)
2. = δ(q3, 100)
3. = δ(q4, 00)
4. = δ(q5, 0)
5. = δ(q5, ε)

As state q5 is the accept state. We get the complete scanned, and we reached to the final state.

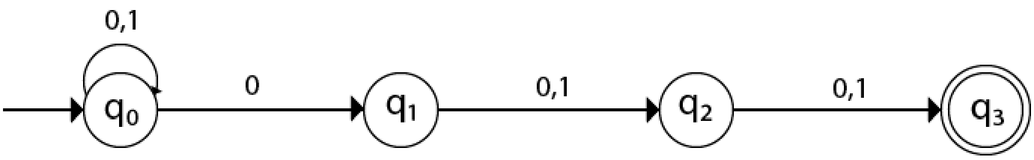
Example 5:

Design an NFA with ∑ = {0, 1} accepts all string in which the third symbol from the right end is always 0.

Solution:



Thus we get the third symbol from the right end as '0' always. The NFA can be:



The above image is an NFA because in state q0 with input 0, we can either go to state q0 or q1.

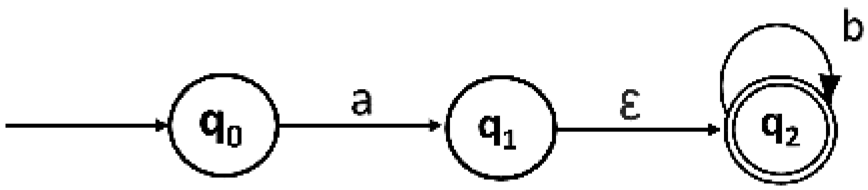
Eliminating ε Transitions

NFA with ε can be converted to NFA without ε, and this NFA without ε can be converted to DFA. To do this, we will use a method, which can remove all the ε transition from given NFA. The method will be:

1. Find out all the ε transitions from each state from Q. That will be called as ε-closure{q1} where qi ∈ Q.
2. Then δ' transitions can be obtained. The δ' transitions mean a ε-closure on δ moves.
3. Repeat Step-2 for each input symbol and each state of given NFA.
4. Using the resultant states, the transition table for equivalent NFA without ε can be built.

Example:

Convert the following NFA with ε to NFA without ε.



Solutions: We will first obtain ε-closures of q0, q1 and q2 as follows:

1. ε-closure(q0) = {q0}
2. ε-closure(q1) = {q1, q2}
3. ε-closure(q2) = {q2}

Now the δ' transition on each input symbol is obtained as:

1. δ'(q0, a) = ε-closure(δ(δ^(q0, ε),a))
2. = ε-closure(δ(ε-closure(q0),a))
3. = ε-closure(δ(q0, a))
4. = ε-closure(q1)
5. = {q1, q2}
6.
7. δ'(q0, b) = ε-closure(δ(δ^(q0, ε),b))
8. = ε-closure(δ(ε-closure(q0),b))
9. = ε-closure(δ(q0, b))

10. $= \Phi$

Now the δ' transition on q1 is obtained as:

1. $\delta'(q1, a) = \varepsilon\text{-closure}(\delta(\delta\hat{}(q1, \varepsilon),a))$
2. $= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q1),a))$
3. $= \varepsilon\text{-closure}(\delta(q1, q2), a)$
4. $= \varepsilon\text{-closure}(\delta(q1, a) \cup \delta(q2, a))$
5. $= \varepsilon\text{-closure}(\Phi \cup \Phi)$
6. $= \Phi$
7.
8. $\delta'(q1, b) = \varepsilon\text{-closure}(\delta(\delta\hat{}(q1, \varepsilon),b))$
9. $= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q1),b))$
10. $= \varepsilon\text{-closure}(\delta(q1, q2), b)$
11. $= \varepsilon\text{-closure}(\delta(q1, b) \cup \delta(q2, b))$
12. $= \varepsilon\text{-closure}(\Phi \cup q2)$
13. $= \{q2\}$

The δ' transition on q2 is obtained as:

1. $\delta'(q2, a) = \varepsilon\text{-closure}(\delta(\delta\hat{}(q2, \varepsilon),a))$
2. $= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q2),a))$
3. $= \varepsilon\text{-closure}(\delta(q2, a))$
4. $= \varepsilon\text{-closure}(\Phi)$
5. $= \Phi$
6.
7. $\delta'(q2, b) = \varepsilon\text{-closure}(\delta(\delta\hat{}(q2, \varepsilon),b))$
8. $= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q2),b))$
9. $= \varepsilon\text{-closure}(\delta(q2, b))$
10. $= \varepsilon\text{-closure}(q2)$
11. $= \{q2\}$

Now we will summarize all the computed δ' transitions:

1. $\delta'(q0, a) = \{q0, q1\}$
2. $\delta'(q0, b) = \Phi$
3. $\delta'(q1, a) = \Phi$
4. $\delta'(q1, b) = \{q2\}$
5. $\delta'(q2, a) = \Phi$

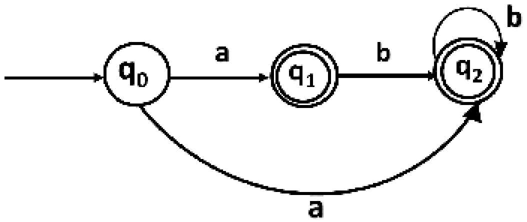
‘

6. δ'(q2, b) = {q2}

The transition table can be:

States	a	b
→q0	{q1, q2}	Φ
*q1	Φ	{q2}
*q2	Φ	{q2}

State q1 and q2 become the final state as ε-closure of q1 and q2 contain the final state q2. The NFA can be shown by the following transition diagram:



Conversion from NFA to DFA

In this section, we will discuss the method of converting NFA to its equivalent DFA. In NFA, when a specific input is given to the current state, the machine goes to multiple states. It can have zero, one or more than one move on a given input symbol. On the other hand, in DFA, when a specific input is given to the current state, the machine goes to only one state. DFA has only one move on a given input symbol.

Let, M = (Q, ∑, δ, q0, F) is an NFA which accepts the language L(M). There should be equivalent DFA denoted by M' = (Q', ∑', q0', δ', F') such that L(M) = L(M').

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.

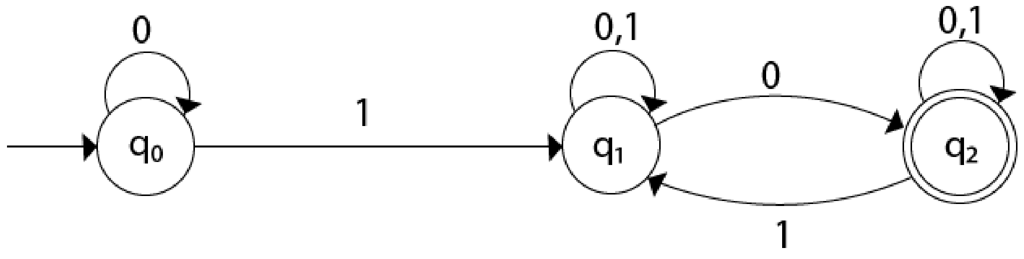
Step 3: In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.

‘

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

Example 1:

Convert the given NFA to DFA.



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

State	0	1
→q0	q0	q1
q1	{q1, q2}	q1
*q2	q2	{q1, q2}

Now we will obtain δ' transition for state q0.

1. δ'([q0], 0) = [q0]
2. δ'([q0], 1) = [q1]

The δ' transition for state q1 is obtained as:

1. δ'([q1], 0) = [q1, q2]   (**new** state generated)
2. δ'([q1], 1) = [q1]

The δ' transition for state q2 is obtained as:

1. δ'([q2], 0) = [q2]
2. δ'([q2], 1) = [q1, q2]

Now we will obtain δ' transition on [q1, q2].

1. δ'([q1, q2], 0) = δ(q1, 0) ∪ δ(q2, 0)
   = {q1, q2} ∪ {q2}

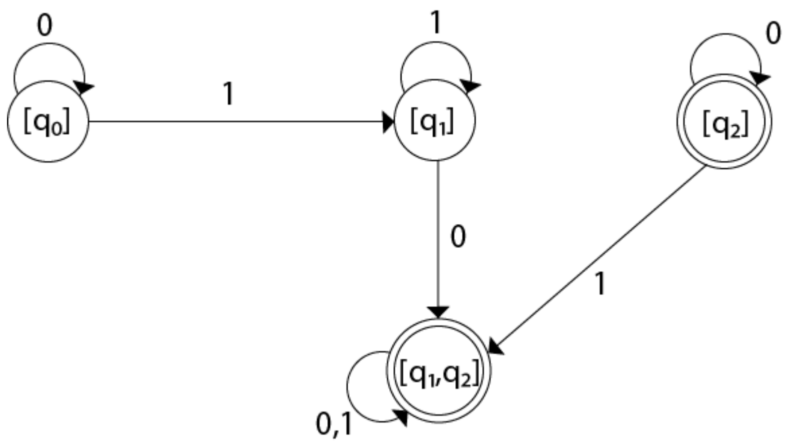
= [q1, q2]

2. δ'([q1, q2], 1) = δ(q1, 1) ∪ δ(q2, 1)
   = {q1} ∪ {q1, q2}
   = {q1, q2}
   = [q1, q2]

The state [q1, q2] is the final state as well because it contains a final state q2. The transition table for the constructed DFA will be:

State	**0**	**1**
→[q0]	[q0]	[q1]
[q1]	[q1, q2]	[q1]
*[q2]	[q2]	[q1, q2]
*[q1, q2]	[q1, q2]	[q1, q2]

The Transition diagram will be:

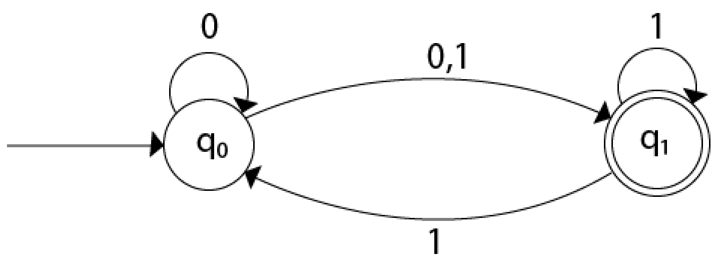


The state q2 can be eliminated because q2 is an unreachable state.

Example 2:

Convert the given NFA to DFA.

‘



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

State	0	1
→q0	{q0, q1}	{q1}
*q1	ϕ	{q0, q1}

Now we will obtain δ' transition for state q0.

1. δ'([q0], 0) = {q0, q1}
2. = [q0, q1] (**new** state generated)
3. δ'([q0], 1) = {q1} = [q1]

The δ' transition for state q1 is obtained as:

1. δ'([q1], 0) = ϕ
2. δ'([q1], 1) = [q0, q1]

Now we will obtain δ' transition on [q0, q1].

1. δ'([q0, q1], 0) = δ(q0, 0) ∪ δ(q1, 0)
2. = {q0, q1} ∪ ϕ
3. = {q0, q1}
4. = [q0, q1]

Similarly,

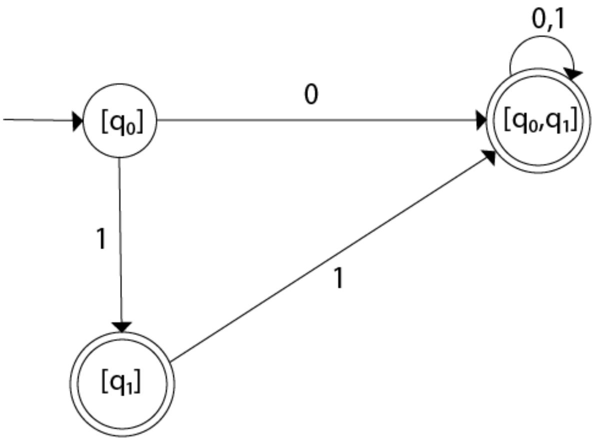
1. δ'([q0, q1], 1) = δ(q0, 1) ∪ δ(q1, 1)
2. = {q1} ∪ {q0, q1}
3. = {q0, q1}
4. = [q0, q1]

As in the given NFA, q1 is a final state, then in DFA wherever, q1 exists that state becomes a final state. Hence in the DFA, final states are [q1] and [q0, q1]. Therefore set of final states F = {[q1], [q0, q1]}.

The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0, q1]	[q1]
*[q1]	ϕ	[q0, q1]
*[q0, q1]	[q0, q1]	[q0, q1]

The Transition diagram will be:



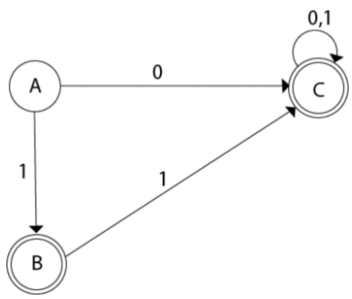
Even we can change the name of the states of DFA.

Suppose

1. A = [q0]
2. B = [q1]
3. C = [q0, q1]

With these new names the DFA will be as follows:

‘



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

ε-closure: ε-closure for a given state A means a set of states which can be reached from the state A with only ε(null) move including the state A itself.

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.

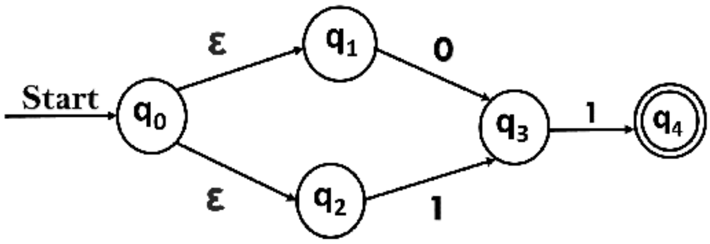
Step 4: Repeat Step 2 and Step 3 until there is no new state present in the transition table of DFA.

‘

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

Example 1:

Convert the NFA with ε into its equivalent DFA.



Solution:

Let us obtain ε-closure of each state.

1. ε-closure {q0} = {q0, q1, q2}
2. ε-closure {q1} = {q1}
3. ε-closure {q2} = {q2}
4. ε-closure {q3} = {q3}
5. ε-closure {q4} = {q4}

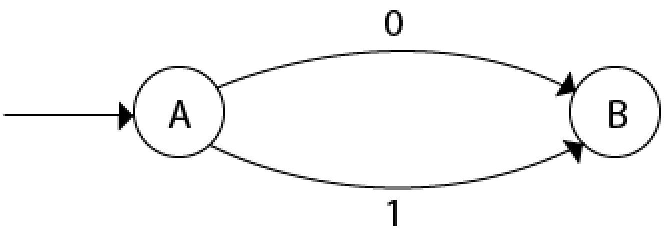
Now, let ε-closure {q0} = {q0, q1, q2} be state A.

Hence

```
δ'(A, 0) = ε-closure {δ((q0, q1, q2), 0) }
            = ε-closure {δ(q0, 0) ∪ δ(q1, 0) ∪ δ(q2, 0) }
            = ε-closure {q3}
            = {q3}                    call it as state B.

δ'(A, 1) = ε-closure {δ((q0, q1, q2), 1) }
            = ε-closure {δ((q0, 1) ∪ δ(q1, 1) ∪ δ(q2, 1) }
            = ε-closure {q3}
            = {q3} = B.
```

The partial DFA will be

‘



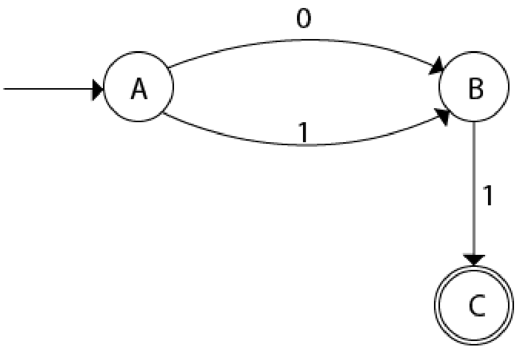
Now,

```
δ'(B, 0) = ε-closure {δ(q3, 0) }
                = ϕ
δ'(B, 1) = ε-closure {δ(q3, 1) }
                = ε-closure {q4}
                = {q4}
```
i.e. state C

For state C:

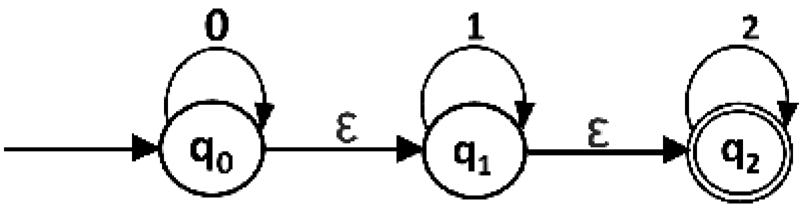
1. δ'(C, 0) = ε-closure {δ(q4, 0) }
2. = ϕ
3. δ'(C, 1) = ε-closure {δ(q4, 1) }
4. = ϕ

The DFA will be,



Example 2:

Convert the given NFA into its equivalent DFA.



‘

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

1. ε-closure(q0) = {q0, q1, q2}
2. ε-closure(q1) = {q1, q2}
3. ε-closure(q2) = {q2}

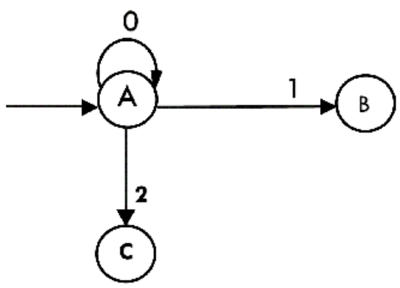
Now we will obtain δ' transition. Let ε-closure(q0) = {q0, q1, q2} call it as **state A**.

```
δ'(A, 0) = ε-closure{δ((q0, q1, q2), 0)}
              = ε-closure{δ(q0, 0) ∪ δ(q1, 0) ∪ δ(q2, 0)}
              = ε-closure{q0}
              = {q0, q1, q2}

δ'(A, 1) = ε-closure{δ((q0, q1, q2), 1)}
              = ε-closure{δ(q0, 1) ∪ δ(q1, 1) ∪ δ(q2, 1)}
              = ε-closure{q1}
              = {q1, q2}          call it as state B

δ'(A, 2) = ε-closure{δ((q0, q1, q2), 2)}
              = ε-closure{δ(q0, 2) ∪ δ(q1, 2) ∪ δ(q2, 2)}
              = ε-closure{q2}
              = {q2}          call it state C
```

Thus we have obtained

1. δ'(A, 0) = A
2. δ'(A, 1) = B
3. δ'(A, 2) = C

The partial DFA will be:



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

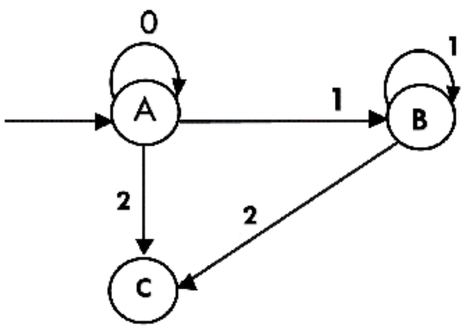
Hence

```
δ'(B, 0) = ε-closure{δ((q1, q2), 0)}
              = ε-closure{δ(q1, 0) ∪ δ(q2, 0)}
              = ε-closure{ϕ}
              = ϕ

δ'(B, 1) = ε-closure{δ((q1, q2), 1)}
              = ε-closure{δ(q1, 1) ∪ δ(q2, 1)}
              = ε-closure{q1}
              = {q1, q2}          i.e. state B itself

δ'(B, 2) = ε-closure{δ((q1, q2), 2)}
              = ε-closure{δ(q1, 2) ∪ δ(q2, 2)}
              = ε-closure{q2}
              = {q2}          i.e. state C itself
```

Thus we have obtained

1. δ'(B, 0) = ϕ
2. δ'(B, 1) = B
3. δ'(B, 2) = C

The partial transition diagram will be

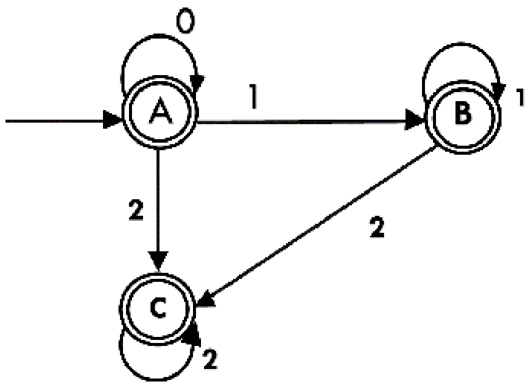


Now we will obtain transitions for C:

```
δ'(C, 0) = ε-closure{δ(q2, 0)}
              = ε-closure{ϕ}
              = ϕ

δ'(C, 1) = ε-closure{δ(q2, 1)}
              = ε-closure{ϕ}
              = ϕ

δ'(C, 2) = ε-closure{δ(q2, 2)}
              = {q2}
```

‘

Hence the DFA is



As A = {q0, q1, q2} in which final state q2 lies hence A is final state. B = {q1, q2} in which the state q2 lies hence B is also final state. C = {q2}, the state q2 lies hence C is also a final state.

Minimization of DFA

Minimization of DFA means reducing the number of states from given FA. Thus, we get the FSM(finite state machine) with redundant states after minimizing the FSM.

We have to follow the various steps to minimize the DFA. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

Step 4: Find similar rows from T1 such that:

1. 1. δ (q, a) = p
2. 2. δ (r, a) = p

That means, find the two states which have the same value of a and b and remove one of them.

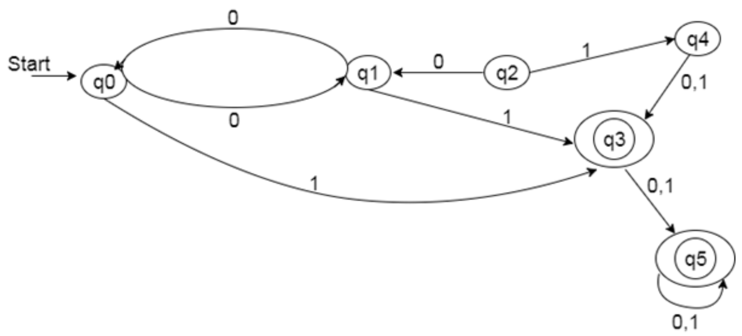
Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

‘

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

Example:



Solution:

Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for the rest of the states.

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q5	q5
*q5	q5	q5

Step 3: Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final states:

State	0	1
q0	q1	q3
q1	q0	q3

2. Another set contains those rows, which starts from final states.

‘

State	0	1
q3	q5	q5
q5	q5	q5

Step 4: Set 1 has no similar rows so set 1 will be the same.

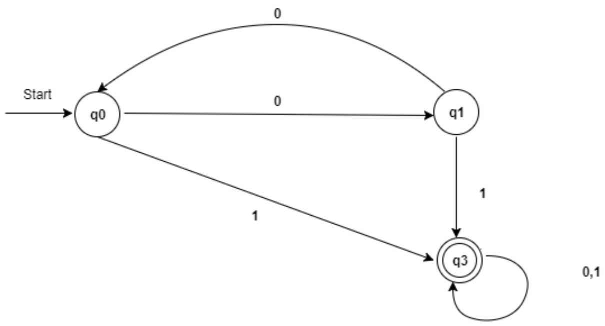
Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

State	0	1
q3	q3	q3

Step 6: Now combine set 1 and set 2 as:

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q3	q3

Now it is the transition table of minimized DFA.



Regular Expression

‘

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

For instance:

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

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

Operations on Regular Language

The various operations on regular language are:

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

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

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

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

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

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

Example 1:

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

Solution:

‘

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

1. R = a*

That is Kleen closure of a.

Example 2:

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

Solution:

The regular expression has to be built for the language

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

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

```
R = a⁺
```

Example 3:

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

Solution:

The regular expression will be:

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

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

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

Examples of Regular Expression

Example 1:

‘

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

Solution:

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

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

Example 2:

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

Solution:

The regular expression will be:

1. R = a b* b

Example 3:

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

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

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

The regular expression for the above language is:

1. R = {a + ab}*

Example 4:

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

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

1. R = a* b* c*

‘

Example 5:

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

Solution:

The regular expression has to be built for the language:

1. L = {ε, 00, 0000, 000000, ...}

The regular expression for the above language is:

1. R = (00)*

Example 6:

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

Solution:

The regular expression will be:

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

Example 7:

Describe the language denoted by following regular expression

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

Solution:

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

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

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

Example 8:

‘

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

Solution:

The Language is as follows:

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

The regular expression for the above language is as follows:

1. R = (1* 0*)

Example 9:

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

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

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

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

Example 10:

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

Solution:

The regular expectation will be:

1. R = (011 + 1)*

Conversion of RE to FA

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

‘

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

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

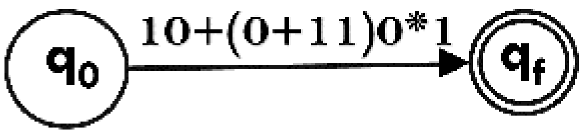
Step 3: Convert the obtained NFA to equivalent DFA.

Example 1:

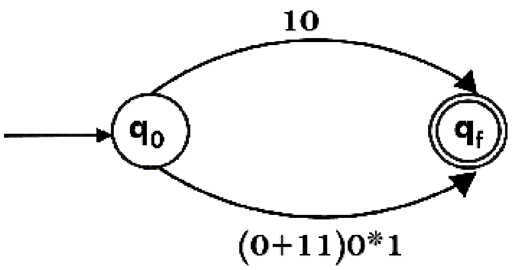
Design a FA from given regular expression 10 + (0 + 11)0* 1.

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

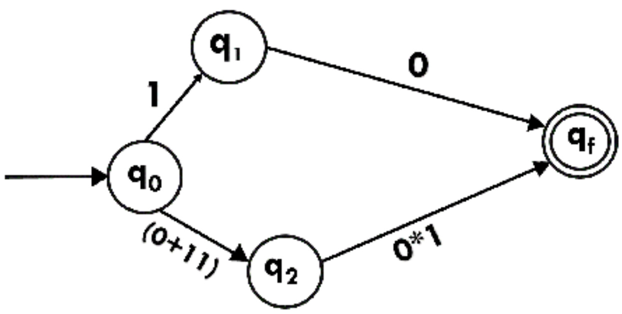
Step 1:



Step 2:

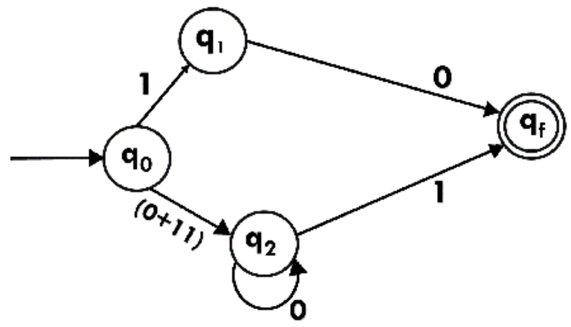


Step 3:

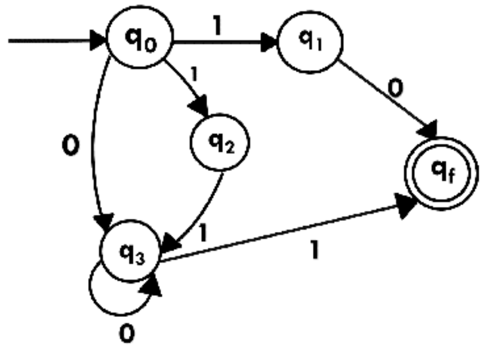


Step 4:

‘



Step 5:



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

State	0	1
→q0	q3	{q1, q2}
q1	qf	ϕ
q2	ϕ	q3
q3	q3	qf
*qf	ϕ	ϕ

The equivalent DFA will be:

State	0	1
→[q0]	[q3]	[q1, q2]

‘

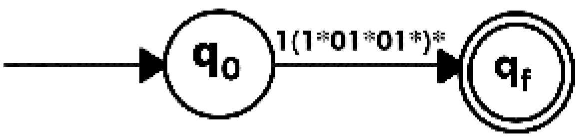
[q1]	[qf]	ϕ
[q2]	ϕ	[q3]
[q3]	[q3]	[qf]
[q1, q2]	[qf]	[qf]
*[qf]	ϕ	ϕ

Example 2:

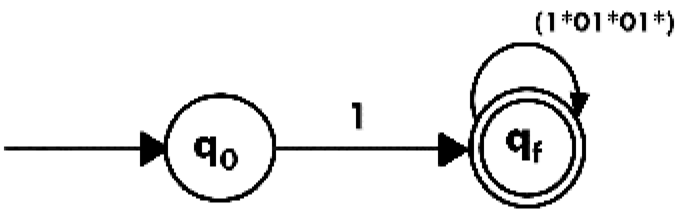
Design a NFA from given regular expression 1 (1* 01* 01*)*.

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

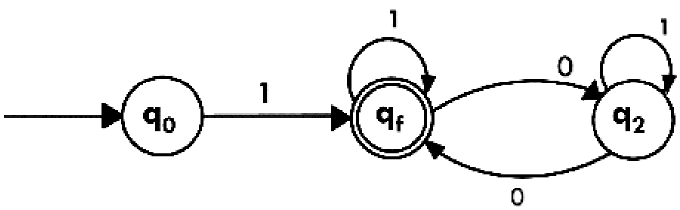
Step 1:



Step 2:



Step 3:



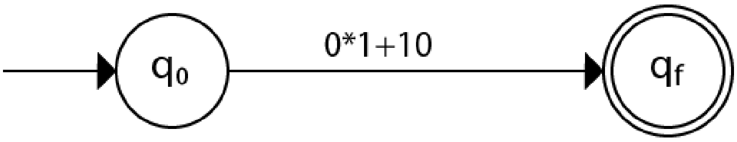
Example 3:

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

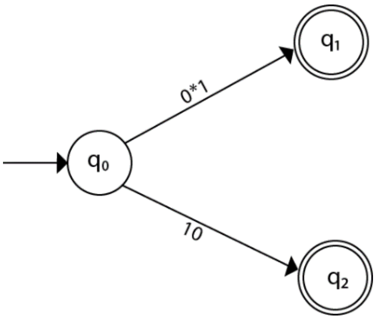
Solution:

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

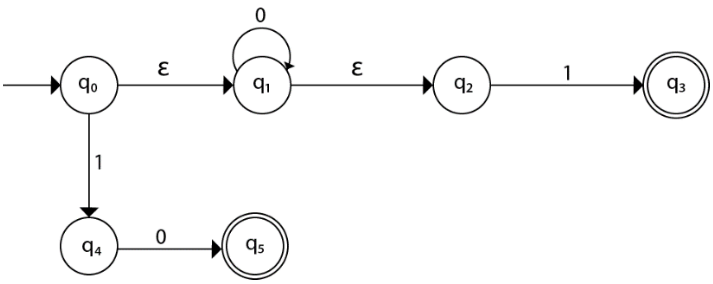
Step 1:



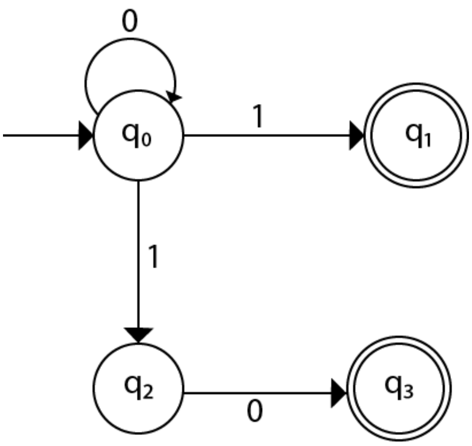
Step 2:



Step 3:



Step 4:



‘

Theorem

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

$|w| \geq c$

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

- $|y| > 0$
- $|xy| \leq c$
- For all $k \geq 0$, the string xy^kz 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 **$|w| \geq c$**
  - Select **y** such that **$|y| \geq 1$**
  - Select **x** such that **$|xy| \leq c$**
  - Assign the remaining string to **z.**
  - Select **k** such that the resulting string is not in **L.**

Hence L is not regular.

Problem

Prove that **$L = \{a^ib^i \mid i \geq 0\}$** 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 $|w| = 2n \geq n$.
- By pumping lemma, let w = xyz, where $|xy| \leq n$.
- Let $x = a^p$, $y = a^q$, and $z = a^rb^n$, where $p + q + r = n$, $p \neq 0$, $q \neq 0$, $r \neq 0$. Thus $|y| \neq 0$.
- Let k = 2. Then $xy^2z = a^pa^{2q}a^rb^n$.

'

- Number of as = (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.

Arden's Theorem

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

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

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

1. Let q_1 be the initial state.

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

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

4. Calculate q_i such that

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

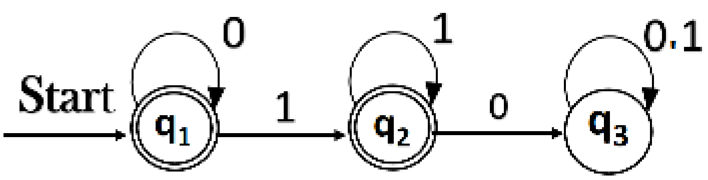
If q_j is a start state then we have:

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

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

Example:

Construct the regular expression for the given DFA



Solution:

‘

Let us write down the equations

```
q1 = q1 0 + ε
```

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

Similarly,

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

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

```
q1 =  q1 0 + ε
```

We can re-write it as

```
q1 = ε + q1 0
```

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

Assuming R = q1, Q = ε, P = 0

We get

```
q1 = ε.(0)*
q1 = 0*    (ε.R*= R*)
```

Substituting the value into q2, we will get

```
q2 = 0* 1 + q2 1
q2 = 0* 1 (1)*   (R = Q + RP  →  Q P*)
```

The regular expression is given by

```
r = q1 + q2
= 0* + 0* 1.1*
r = 0* + 0* 1+    (1.1* = 1+)
```

Moore Machine

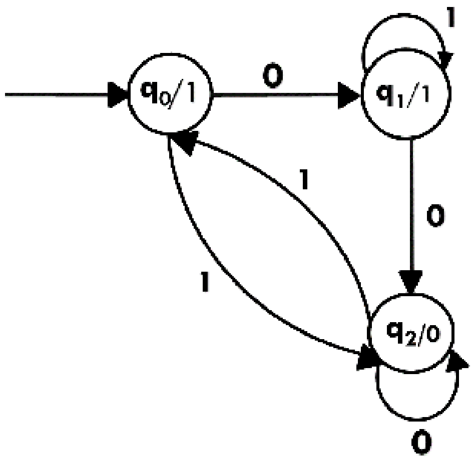
'

Moore machine is a finite state machine in which the next state is decided by the current state and current input symbol. The output symbol at a given time depends only on the present state of the machine. Moore machine can be described by 6 tuples (Q, q0, ∑, O, δ, λ) where,

1. Q: finite set of states
2. q0: initial state of machine
3. ∑: finite set of input symbols
4. O: output alphabet
5. δ: transition function where Q × ∑ → Q
6. λ: output function where Q → O

Example 1:

The state diagram for Moore Machine is



Transition table for Moore Machine is:

Current State	Next State (δ)		Output(λ)
	0	1	
q_0	q_1	q_2	1
q_1	q_2	q_1	1
q_2	q_2	q_0	0

‘

In the above Moore machine, the output is represented with each input state separated by /. The output length for a Moore machine is greater than input by 1.

Input: 010

Transition: δ (q0,0) => δ(q1,1) => δ(q1,0) => q2

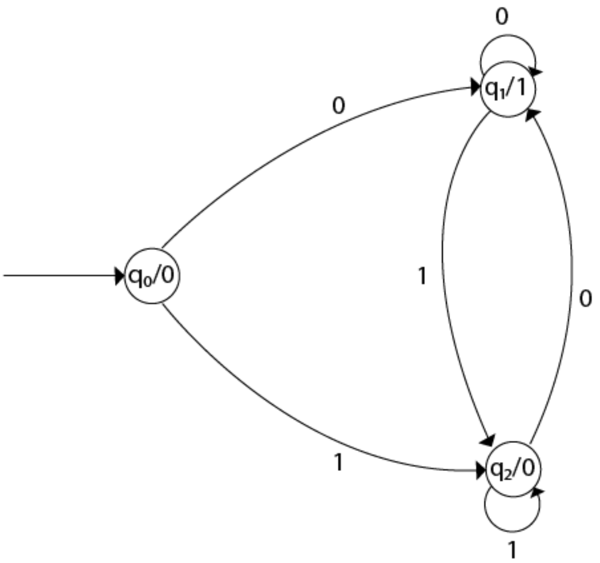
Output: 1110(1 for q0, 1 for q1, again 1 for q1, 0 for q2)

Example 2:

Design a Moore machine to generate 1's complement of a given binary number.

Solution: To generate 1's complement of a given binary number the simple logic is that if the input is 0 then the output will be 1 and if the input is 1 then the output will be 0. That means there are three states. One state is start state. The second state is for taking 0's as input and produces output as 1. The third state is for taking 1's as input and producing output as 0.

Hence the Moore machine will be,



For instance, take one binary number 1011 then

Input		1	0	1	1
State	q0	q2	q1	q2	q2
Output	0	0	1	0	0

‘

Thus we get 00100 as 1's complement of 1011, we can neglect the initial 0 and the output which we get is 0100 which is 1's complement of 1011. The transaction table is as follows:

	δ		λ
Current State	Next State		Output
	0	1	
→ q_0	q_1	q_2	0
q_1	q_1	q_2	1
q_2	q_1	q_2	0

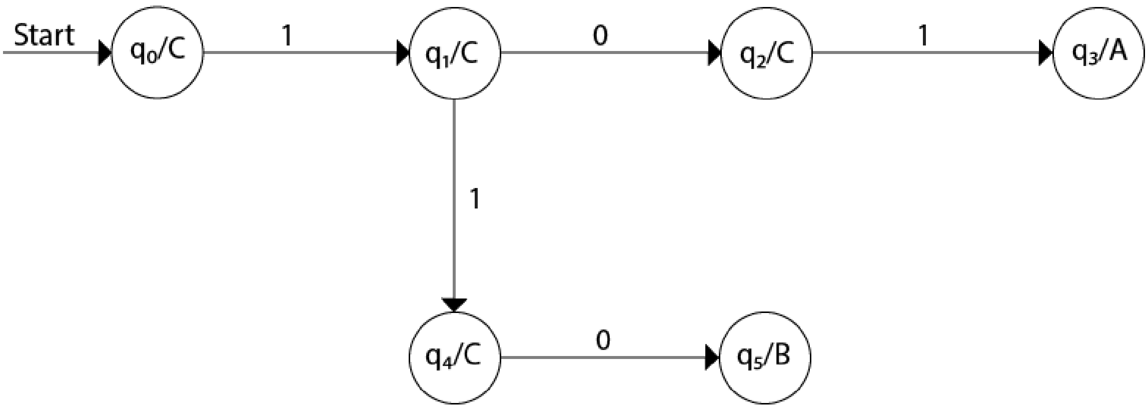
Thus Moore machine M = (Q, q0, ∑, O, δ, λ); where Q = {q0, q1, q2}, ∑ = {0, 1}, O = {0, 1}. the transition table shows the δ and λ functions.

Example 3:

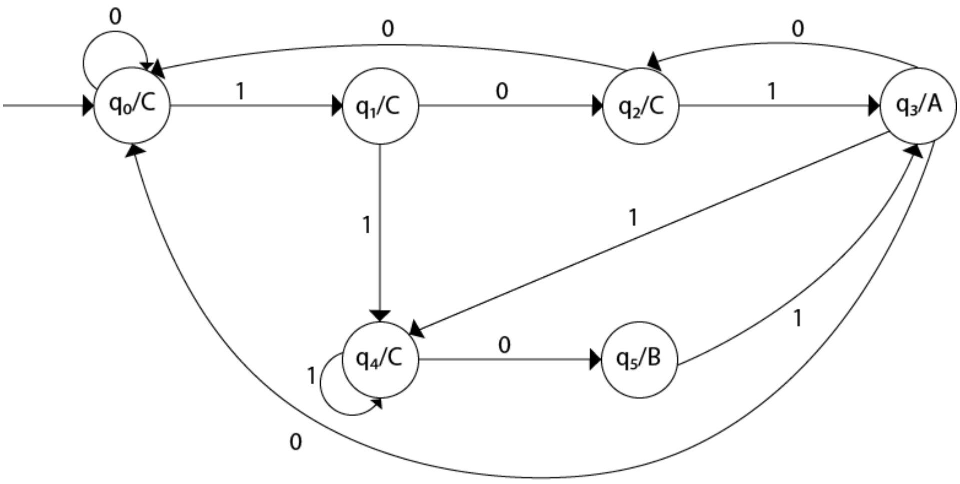
Design a Moore machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B otherwise it outputs C.

Solution: For designing such a machine, we will check two conditions, and those are 101 and 110. If we get 101, the output will be A, and if we recognize 110, the output will be B. For other strings, the output will be C.

The partial diagram will be:



Now we will insert the possibilities of 0's and 1's for each state. Thus the Moore machine becomes:

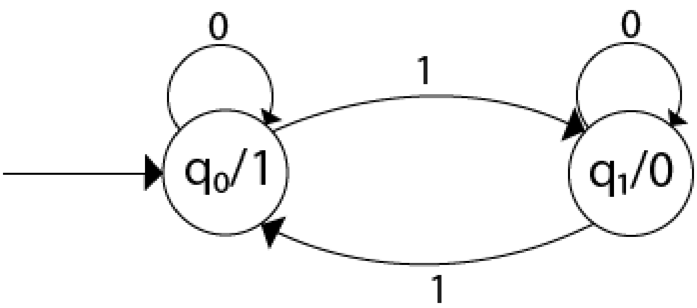


Example 4:

Construct a Moore machine that determines whether an input string contains an even or odd number of 1's. The machine should give 1 as output if an even number of 1's are in the string and 0 otherwise.

Solution:

The Moore machine will be:



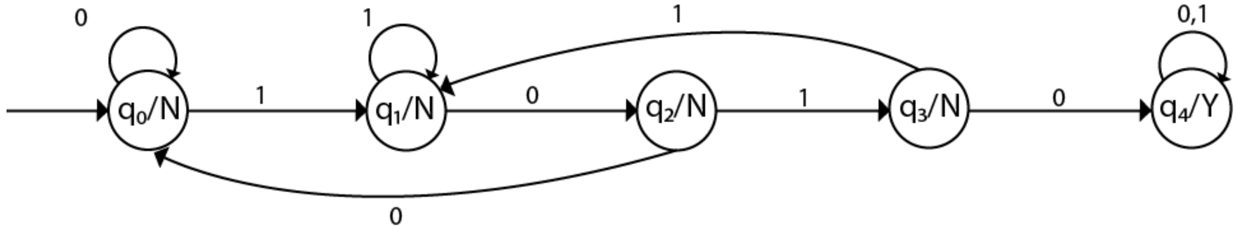
This is the required Moore machine. In this machine, state q1 accepts an odd number of 1's and state q0 accepts even number of 1's. There is no restriction on a number of zeros. Hence for 0 input, self-loop can be applied on both the states.

Example 5:

Design a Moore machine with the input alphabet {0, 1} and output alphabet {Y, N} which produces Y as output if input sequence contains 1010 as a substring otherwise, it produces N as output.

Solution:

The Moore machine will be:



Mealy Machine

A Mealy machine is a machine in which output symbol depends upon the present input symbol and present state of the machine. In the Mealy machine, the output is represented with each input symbol for each state separated by /. The Mealy machine can be described by 6 tuples (Q, q0, ∑, O, δ, λ') where

1. Q: finite set of states
2. q0: initial state of machine
3. ∑: finite set of input alphabet
4. O: output alphabet
5. δ: transition function where Q × ∑ → Q
6. λ': output function where Q × ∑ →O

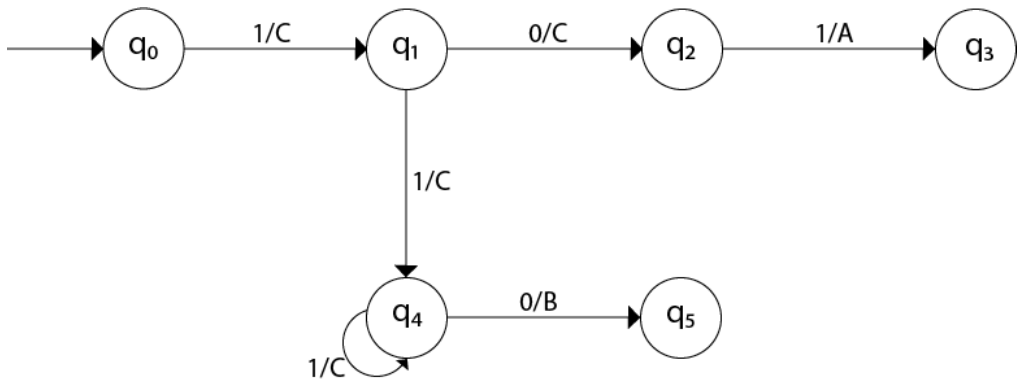
Example 1:

Design a Mealy machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B otherwise it outputs C.

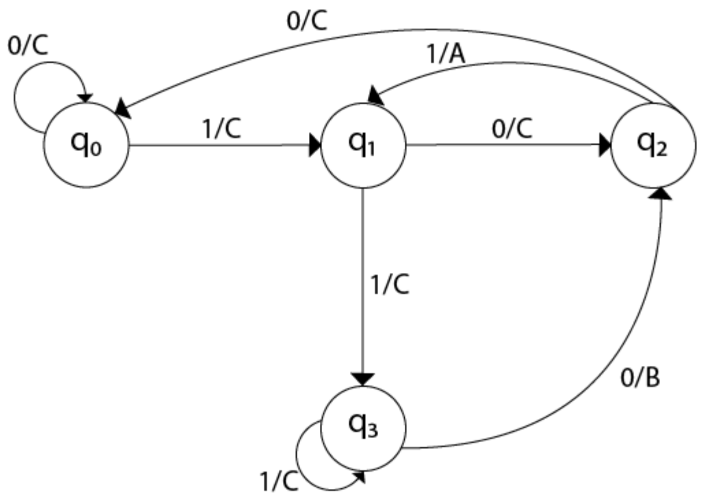
Solution: For designing such a machine, we will check two conditions, and those are 101 and 110. If we get 101, the output will be A. If we recognize 110, the output will be B. For other strings the output will be C.

The partial diagram will be:

‘



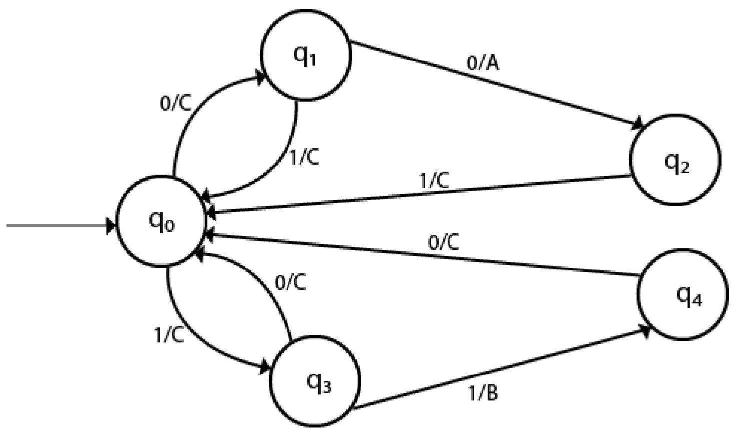
Now we will insert the possibilities of 0's and 1's for each state. Thus the Mealy machine becomes:



Example 2:

Design a mealy machine that scans sequence of input of 0 and 1 and generates output 'A' if the input string terminates in 00, output 'B' if the string terminates in 11, and output 'C' otherwise.

Solution: The mealy machine will be:



‘

Conversion from Mealy machine to Moore Machine

In Moore machine, the output is associated with every state, and in Mealy machine, the output is given along the edge with input symbol. To convert Moore machine to Mealy machine, state output symbols are distributed to input symbol paths. But while converting the Mealy machine to Moore machine, we will create a separate state for every new output symbol and according to incoming and outgoing edges are distributed.

The following steps are used for converting Mealy machine to the Moore machine:

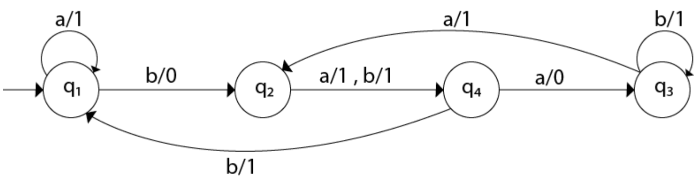
Step 1: For each state(Qi), calculate the number of different outputs that are available in the transition table of the Mealy machine.

Step 2: Copy state Qi, if all the outputs of Qi are the same. Break qi into n states as Qin, if it has n distinct outputs where n = 0, 1, 2....

Step 3: If the output of initial state is 0, insert a new initial state at the starting which gives 1 output.

Example 1:

Convert the following Mealy machine into equivalent Moore machine.



Solution:

Transition table for above Mealy machine is as follows:

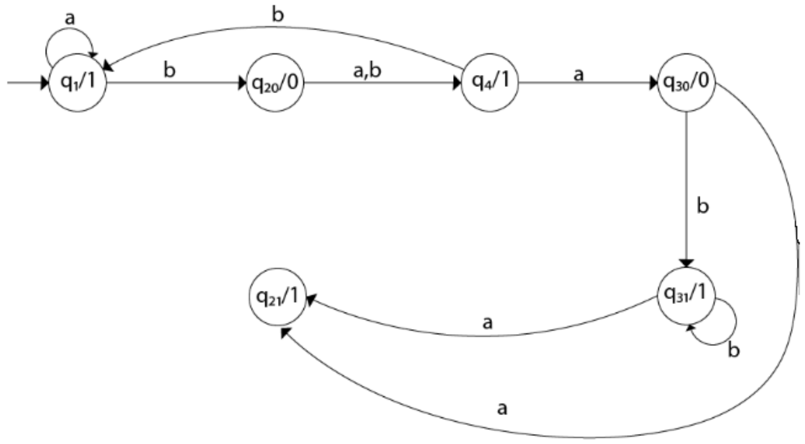
Present State	Next State			
	a		b	
	State	O/P	State	O/P
q_1	q_1	1	q_2	0
q_2	q_4	1	q_4	1
q_3	q_2	1	q_3	1
q_4	q_3	0	q_1	1

- For state q1, there is only one incident edge with output 0. So, we don't need to split this state in Moore machine.
- For state q2, there is 2 incident edge with output 0 and 1. So, we will split this state into two states q20(state with output 0) and q21(with output 1).
- For state q3, there is 2 incident edge with output 0 and 1. So, we will split this state into two states q30(state with output 0) and q31(state with output 1).
- For state q4, there is only one incident edge with output 0. So, we don't need to split this state in Moore machine.

Transition table for Moore machine will be:

Present State	Next State		Output
	a=0	a=1	
q_1	q_1	q_2	1
q_{20}	q_4	q_4	0
q_{21}	ø	ø	1
q_{30}	q_{21}	q_{31}	0
q_{31}	q_{21}	q_{31}	1
q_4	q_3	q_4	1

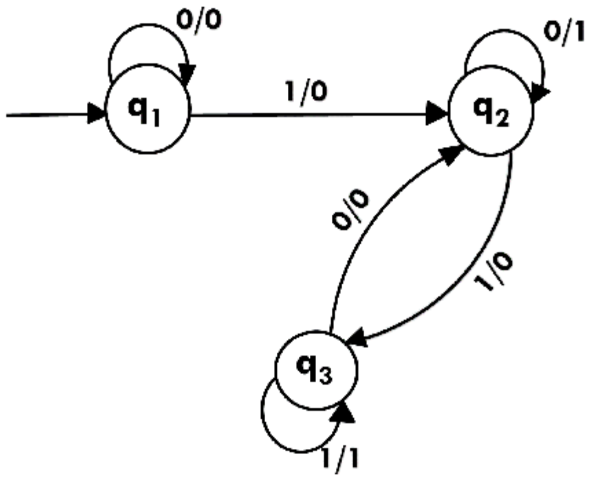
Transition diagram for Moore machine will be:



Example 2:

Convert the following Mealy machine into equivalent Moore machine.

‘



Solution:

Transition table for above Mealy machine is as follows:

Present State	Next State 0		Next State 1	
	State	o/P	State	o/P
q_1	q_1	0	q_2	0
q_2	q_2	1	q_3	0
q_3	q_2	0	q_3	1

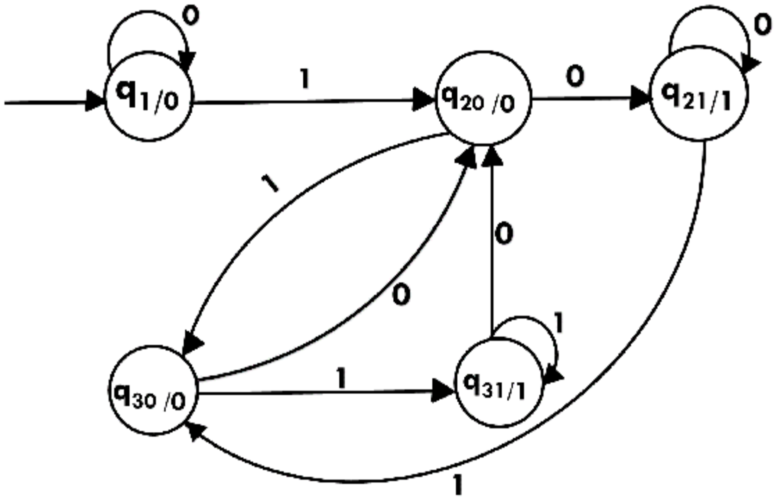
The state q1 has only one output. The state q2 and q3 have both output 0 and 1. So we will create two states for these states. For q2, two states will be q20(with output 0) and q21(with output 1). Similarly, for q3 two states will be q30(with output 0) and q31(with output 1).

Transition table for Moore machine will be:

Present State	Next State 0	Next State 1	o/P
q_1	q_1	q_{20}	0
q_{20}	q_{21}	q_{30}	0
q_{21}	q_{21}	q_{30}	1
q_{30}	q_{20}	q_{31}	0
q_{31}	q_{20}	q_{31}	1

‘

Transition diagram for Moore machine will be:



Conversion from Moore machine to Mealy Machine

In the Moore machine, the output is associated with every state, and in the mealy machine, the output is given along the edge with input symbol. The equivalence of the Moore machine and Mealy machine means both the machines generate the same output string for same input string.

We cannot directly convert Moore machine to its equivalent Mealy machine because the length of the Moore machine is one longer than the Mealy machine for the given input. To convert Moore machine to Mealy machine, state output symbols are distributed into input symbol paths. We are going to use the following method to convert the Moore machine to Mealy machine.

Method for conversion of Moore machine to Mealy machine

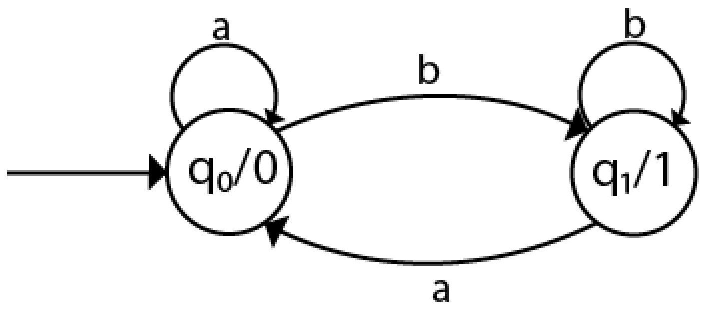
Let M = (Q, ∑, δ, λ, q0) be a Moore machine. The equivalent Mealy machine can be represented by M' = (Q, ∑, δ, λ', q0). The output function λ' can be obtained as:

1. λ' (q, a) = λ(δ(q, a))

Example 1:

Convert the following Moore machine into its equivalent Mealy machine.

‘



Solution:

The transition table of given Moore machine is as follows:

Q	a	b	Output(λ)
q0	q0	q1	0
q1	q0	q1	1

The equivalent Mealy machine can be obtained as follows:

1. λ' (q0, a) = λ(δ(q0, a))
2. = λ(q0)
3. = 0
4.
5. λ' (q0, b) = λ(δ(q0, b))
6. = λ(q1)
7. = 1

The λ for state q1 is as follows:

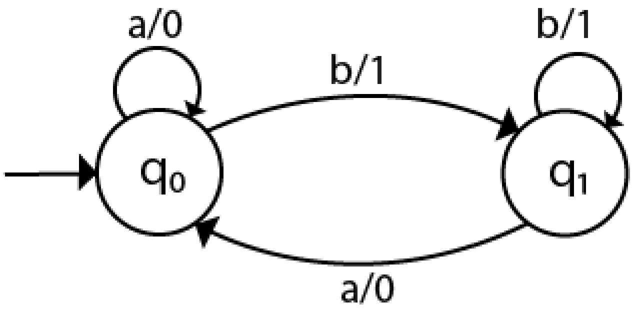
1. λ' (q1, a) = λ(δ(q1, a))
2. = λ(q0)
3. = 0
4.
5. λ' (q1, b) = λ(δ(q1, b))
6. = λ(q1)
7. = 1

Hence the transition table for the Mealy machine can be drawn as follows:

‘

Q \ Σ	Input 0		Input 1	
	State	**O/P**	**State**	**O/P**
q_0	q_0	0	q_1	1
q_1	q_0	0	q_1	1

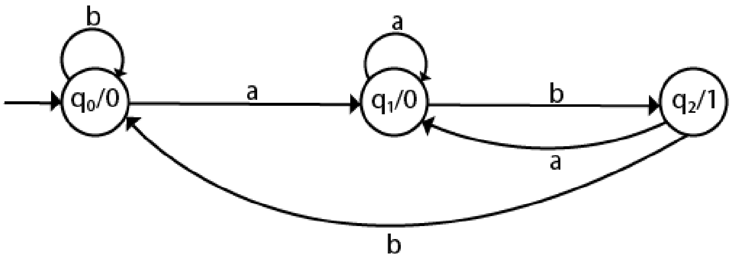
The equivalent Mealy machine will be,



Note: The length of output sequence is 'n+1' in Moore machine and is 'n' in the Mealy machine.

Example 2:

Convert the given Moore machine into its equivalent Mealy machine.



Solution:

The transition table of given Moore machine is as follows:

Q	a	b	Output(λ)
q0	q1	q0	0
q1	q1	q2	0

q2	q1	q0	1

The equivalent Mealy machine can be obtained as follows:

1. λ' (q0, a) = λ(δ(q0, a))
2. = λ(q1)
3. = 0
4.
5. λ' (q0, b) = λ(δ(q0, b))
6. = λ(q0)
7. = 0

The λ for state q1 is as follows:

1. λ' (q1, a) = λ(δ(q1, a))
2. = λ(q1)
3. = 0
4.
5. λ' (q1, b) = λ(δ(q1, b))
6. = λ(q2)
7. = 1

The λ for state q2 is as follows:

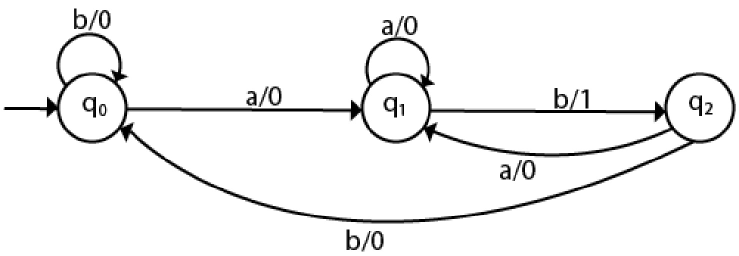
1. λ' (q2, a) = λ(δ(q2, a))
2. = λ(q1)
3. = 0
4.
5. λ' (q2, b) = λ(δ(q2, b))
6. = λ(q0)
7. = 0

Hence the transition table for the Mealy machine can be drawn as follows:

‘

Q \ Σ	Input a		Input b	
	State	Output	State	Output
q_0	q_1	0	q_0	0
q_1	q_1	0	q_2	1
q_2	q_1	0	q_0	0

The equivalent Mealy machine will be,



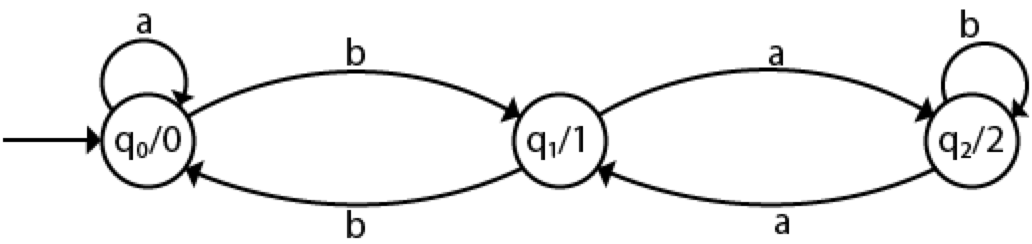
Example 3:

Convert the given Moore machine into its equivalent Mealy machine.

Q	a	b	Output(λ)
q0	q0	q1	0
q1	q2	q0	1
q2	q1	q2	2

Solution:

The transaction diagram for the given problem can be drawn as:



The equivalent Mealy machine can be obtained as follows:

‘

$$\lambda'(q0, a) = \lambda(\delta(q0, a))$$
$$= \lambda(q0)$$
$$= 0$$

$$\lambda'(q0, b) = \lambda(\delta(q0, b))$$
$$= \lambda(q1)$$
$$= 1$$

The λ for state q1 is as follows:

$$\lambda'(q1, a) = \lambda(\delta(q1, a))$$
$$= \lambda(q2)$$
$$= 2$$

$$\lambda'(q1, b) = \lambda(\delta(q1, b))$$
$$= \lambda(q0)$$
$$= 0$$

The λ for state q2 is as follows:

$$\lambda'(q2, a) = \lambda(\delta(q2, a))$$
$$= \lambda(q1)$$
$$= 1$$

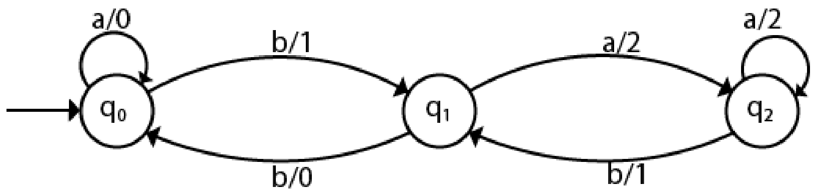
$$\lambda'(q2, b) = \lambda(\delta(q2, b))$$
$$= \lambda(q2)$$
$$= 2$$

Hence the transition table for the Mealy machine can be drawn as follows:

Q \ Σ	Input a		Input b	
	State	**O/P**	**State**	**O/P**
q_0	q_0	0	q_1	1
q_1	q_2	2	q_0	0
q_2	q_1	1	q_2	2

The equivalent Mealy machine will be,

‘



Context-Free Grammar (CFG)

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

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

Where,

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

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

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

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

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

Example 1:

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

‘

Solution:

As we know the regular expression for the above language is

1. r.e. = a*

Production rule for the Regular expression is as follows:

1. S → aS   rule 1
2. S → ε   rule 2

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

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

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

Example 2:

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

Solution:

The CFG can be given by,

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

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

‘

Example 3:

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

Solution:

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

The grammar could be:

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

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

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

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

Example 4:

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

Solution:

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

The grammar could be:

1. S → aSbb | abb

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

1. S → aSbb
2. S → aabbbb

‘

Derivation

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

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

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

1. Leftmost Derivation:

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

Example:

Production rules:

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

Input

1. a - b + a

The leftmost derivation is:

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

2. Rightmost Derivation:

‘

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

Example

Production rules:

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

Input

1. a - b + a

The rightmost derivation is:

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

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

Examples of Derivation:

Example 1:

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

1. S → AB | ε
2. A → aB
3. B → Sb

Solution:

Leftmost derivation:

‘

S

AB

$\boxed{aB}$ B

a $\boxed{Sb}$ B

A $\boxed{\varepsilon}$ bB

ab $\boxed{Sb}$

ab $\boxed{\varepsilon}$ b

abb

Rightmost derivation:

S

AB

A $\boxed{Sb}$

A $\boxed{\varepsilon}$ b

$\boxed{aB}$ b

a $\boxed{Sb}$ b

a $\boxed{\varepsilon}$ bb

abb

Example 2:

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

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

Solution:

Leftmost derivation:

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

Rightmost derivation:

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

Example 3:

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

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

Solution:

Leftmost derivation:

1. S
2. A1B
3. 0A1B
4. 00A1B
5. 001B

‘

6. 0010B
7. 00101B
8. 00101

Rightmost derivation:

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

Derivation Tree

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

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

A parse tree contains the following properties:

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

Example 1:

Production rules:

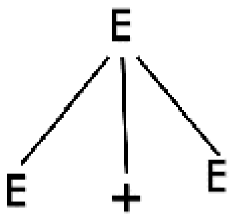
1. E = E + E
2. E = E * E
3. E = a | b | c

‘

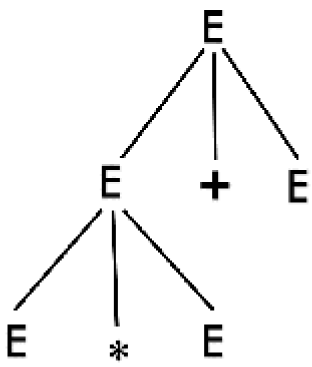
Input

1. a * b + c

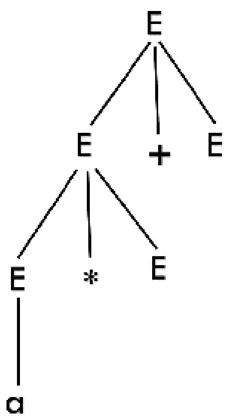
Step 1:



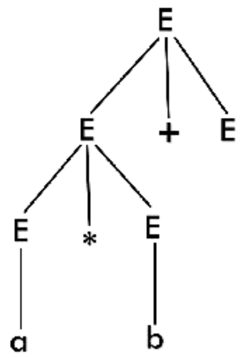
Step 2:



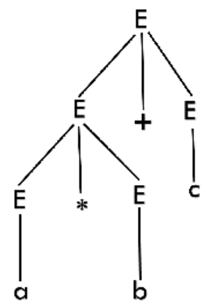
Step 2:



Step 4:



Step 5:



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

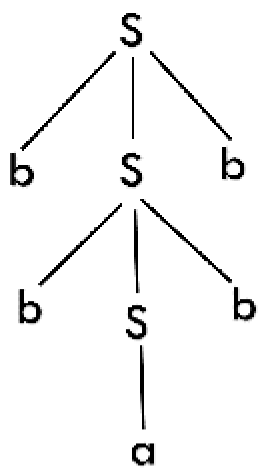
Example 2:

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

1. S → bSb | a | b

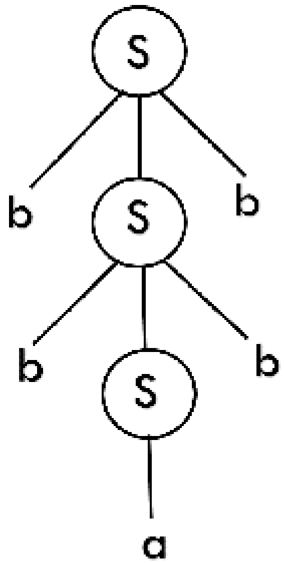
Solution:

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



‘

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



Example 3:

Construct a derivation tree for the string aabbabba for the CFG given by,

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

Solution:

To draw a tree, we will first try to obtain derivation for the string aabbabba

‘

S

aB

a $\boxed{aBB}$

aa $\boxed{bS}$ B

aab $\boxed{bA}$ B

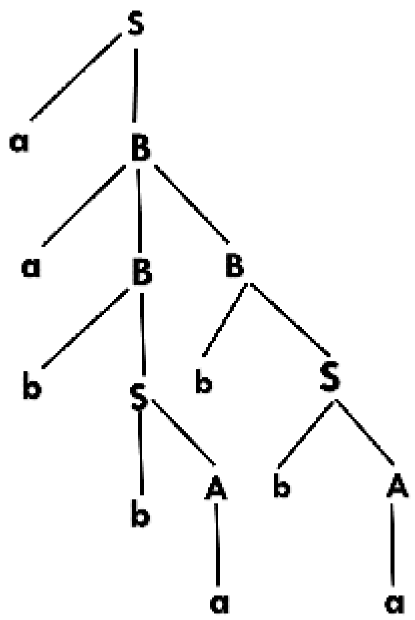
aabb $\boxed{a}$ B

aabba $\boxed{bS}$

aabbab $\boxed{bA}$

aabbabb $\boxed{a}$

Now, the derivation tree is as follows:



Example 4:

Show the derivation tree for string "aabbbb" with the following grammar.

1. S → AB | ε
2. A → aB
3. B → Sb

Solution:

To draw a tree we will first try to obtain derivation for the string aabbbb

S

$\boxed{AB}$

$\boxed{aB}\ B$

$a\ \boxed{Sb}\ B$

$a\ \boxed{AB}\ bB$

$a\ \boxed{aB}\ BbB$

$aa\ \boxed{Sb}\ BbB$

$aa\ \boxed{\varepsilon}\ bBbB$

$aab\ \boxed{Sb}\ bB$

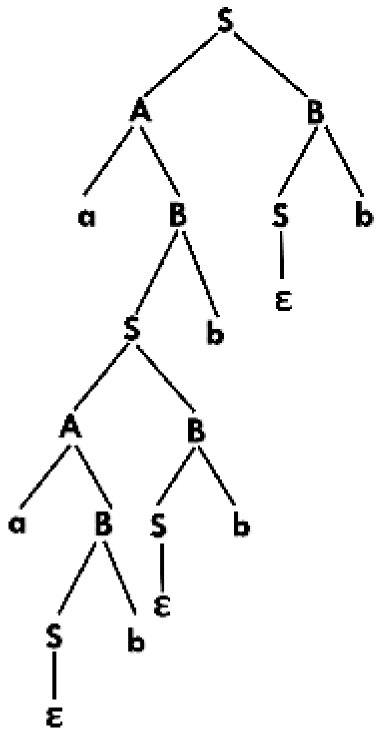
$aab\ \boxed{\varepsilon}\ bbB$

$aabbb\ \boxed{Sb}$

$aabbb\ \boxed{\varepsilon}\ b$

$aabbbb$

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



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.

Example 1:

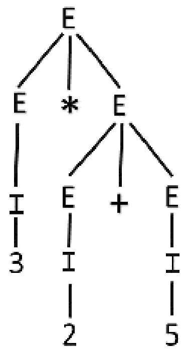
Let us consider a grammar G with the production rule

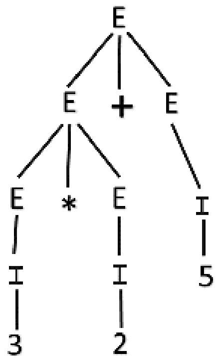
1. E → I
2. E → E + E
3. E → E * E
4. E → (E)
5. I → ε | 0 | 1 | 2 | ... | 9

Solution:

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:

‘





Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is ambiguous.

Example 2:

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.

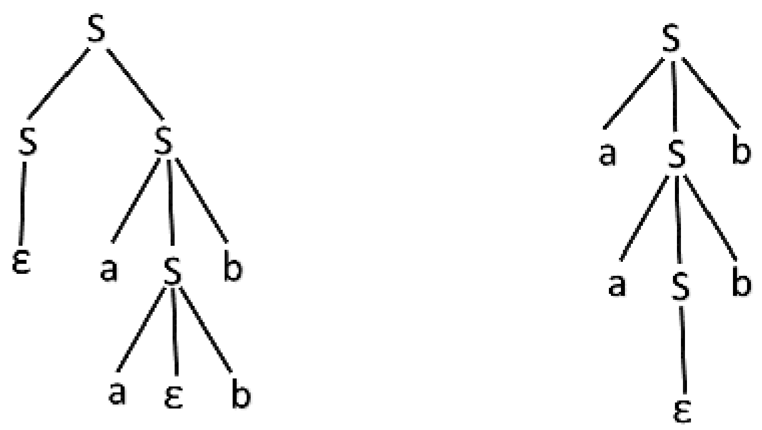
Example 3:

Check whether the given grammar G is ambiguous or not.

1. S → aSb | SS
2. S → ε

Solution:

For the string "aabb" the above grammar can generate two parse trees



Since there are two parse trees for a single string "aabb", the grammar G is ambiguous.

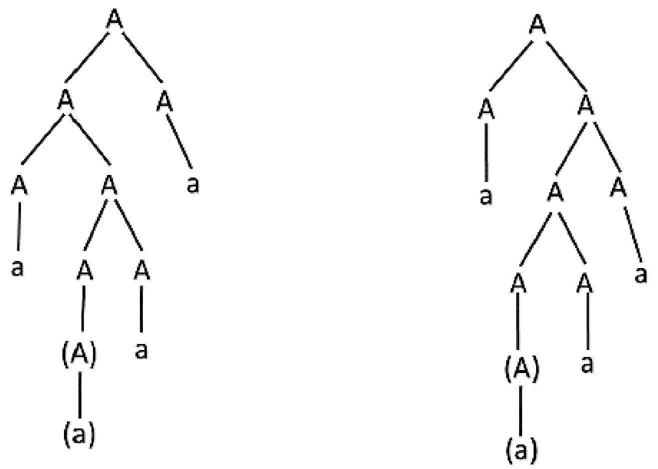
Example 4:

Check whether the given grammar G is ambiguous or not.

1. A → AA
2. A → (A)
3. A → a

Solution:

For the string "a(a)aa" the above grammar can generate two parse trees:



Since there are two parse trees for a single string "a(a)aa", the grammar G is ambiguous.

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,

    1. 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,

    1. X → aX

Example 1:

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

Example 2:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

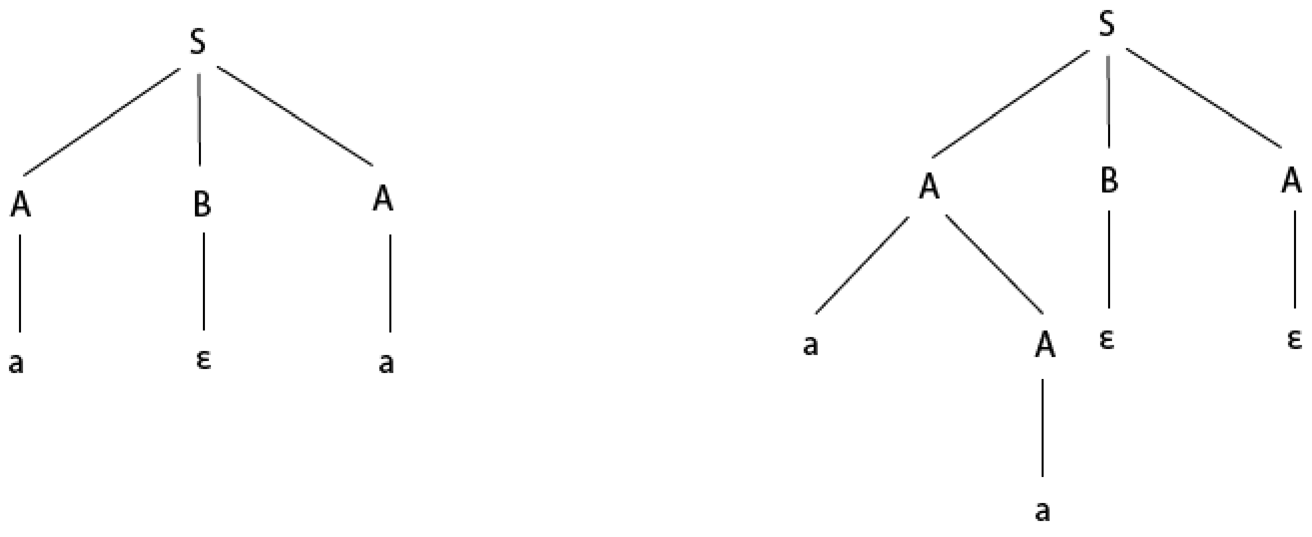
1. S → ABA
2. A → aA | ε

‘

3. B → bB | ε

Solution:

The given grammar is ambiguous because we can derive two different parse tree for string aa.



Parse tree 1 Parse tree 2

The unambiguous grammar is:

1. S → aXY | bYZ | ε
2. Z → aZ | a
3. X → aXY | a | ε
4. Y → bYZ | b | ε

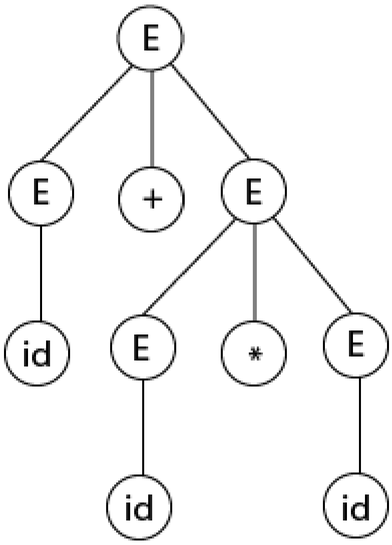
Example 3:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

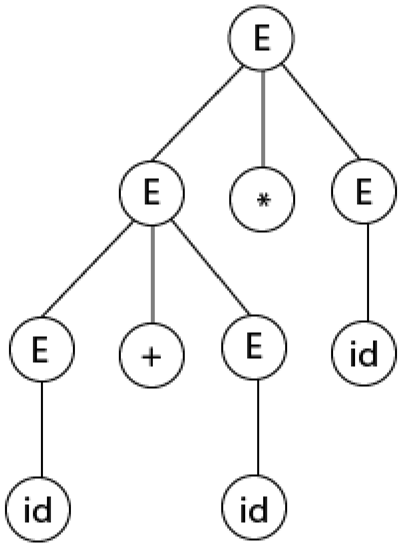
1. E → E + E
2. E → E * E
3. E → id

Solution:

Let us derive the string "id + id * id"



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. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → id

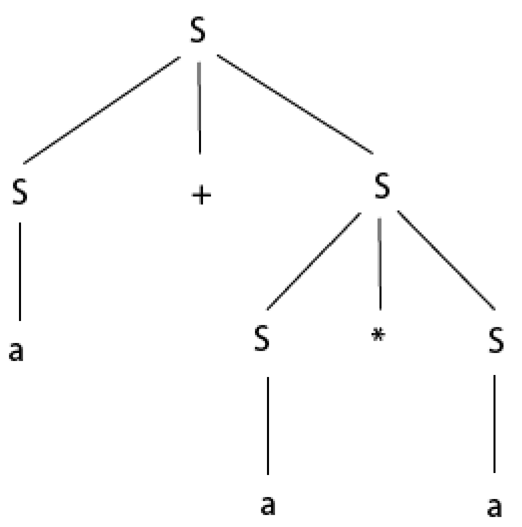
Example 4:

Check that the given grammar is ambiguous or not. Also, find an equivalent unambiguous grammar.

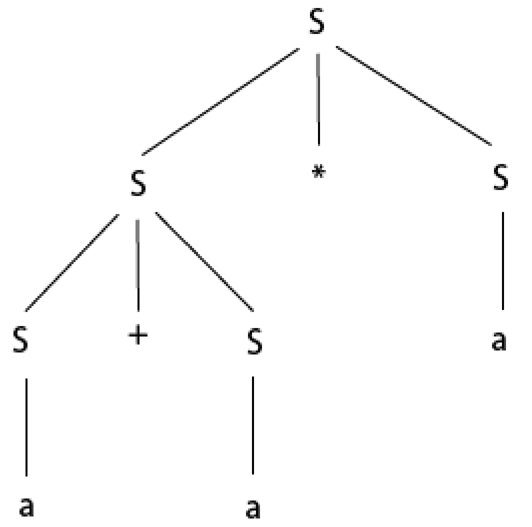
1. S → S + S
2. S → S * S
3. S → S ^ S
4. S → a

Solution:

The given grammar is ambiguous because the derivation of string aab can be represented by the following string:



Parse tree 1



Parse tree 2

Unambiguous grammar will be:

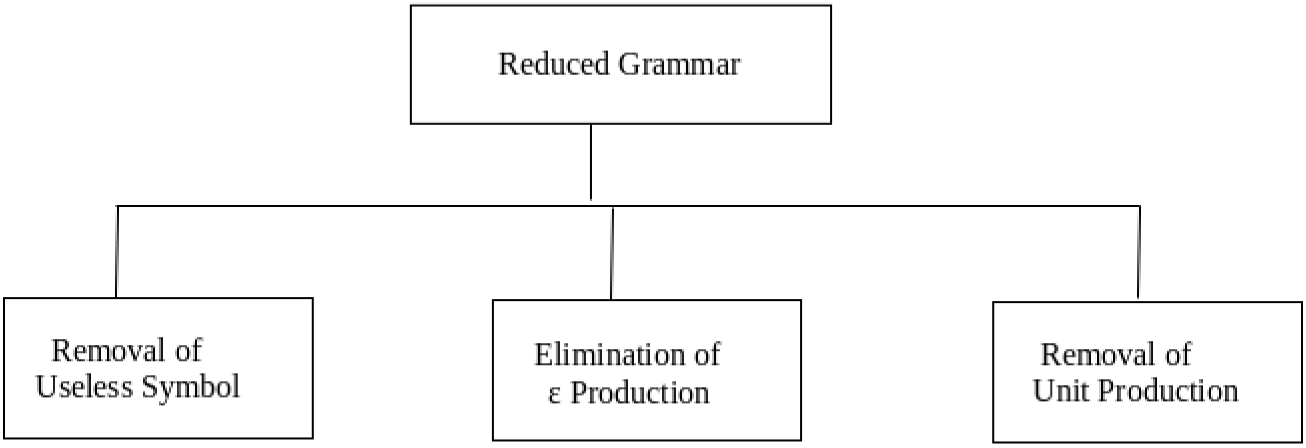
1. S → S + A |
2. A → A * B | B
3. B → C ^ B | C
4. C → a

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). Having extra symbols, unnecessary increase the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols. The properties of reduced grammar are given below:

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

Let us study the reduction process in detail./p>



Removal of Useless Symbols

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

For Example:

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

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

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

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

Elimination of ε Production

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

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

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

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

Example:

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

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

Solution:

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

Let us take

1. S → XYX

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

1. S → YX

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

1. S → XY

If Y = ε then

1. S → XX

If Y and X are ε then,

‘

1. S → X

If both X are replaced by ε

1. S → Y

Now,

1. S → XY | YX | XX | X | Y

Now let us consider

1. X → 0X

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

1. X → 0
2. X → 0X | 0

Similarly Y → 1Y | 1

Collectively we can rewrite the CFG with removed ε production as

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

Removing Unit Productions

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

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

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

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

For example:

1. S → 0A | 1B | C
2. A → 0S | 00

‘

3. B → 1 | A
4. C → 01

Solution:

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

1. S → 0A | 1B | 01

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

1. B → 1 | 0S | 00

Thus finally we can write CFG without unit production as

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

Chomsky's Normal Form (CNF)

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

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

For example:

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

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

‘

Steps for converting CFG into CNF

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

1. S1 → S

Where S1 is the new start symbol.

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

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

1. S → RA
2. R → a

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

1. S → RS
2. R → AS

Example:

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

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

Solution:

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

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

‘

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

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

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

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

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

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

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

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

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

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

‘

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

Greibach Normal Form (GNF)

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

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

For example:

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

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

Steps for converting CFG into GNF

Step 1: Convert the grammar into CNF.

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

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

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

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

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

Example:

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

Solution:

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

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

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

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

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

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

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

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

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

‘

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

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

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

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

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

Pushdown Automata(PDA)

- Pushdown automata is a way to implement a CFG in the same way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.
- Pushdown automata is simply an NFA augmented with an "external stack memory". The addition of stack is used to provide a last-in-first-out memory management capability to Pushdown automata. Pushdown automata can store an unbounded amount of information on the stack. It can access a limited amount of information on the stack. A PDA can push an element onto the top of the stack and pop off an element from the top of the stack. To read an element into the stack, the top elements must be popped off and are lost.
- A PDA is more powerful than FA. Any language which can be acceptable by FA can also be acceptable by PDA. PDA also accepts a class of language which even cannot be accepted by FA. Thus PDA is much more superior to FA.

‘

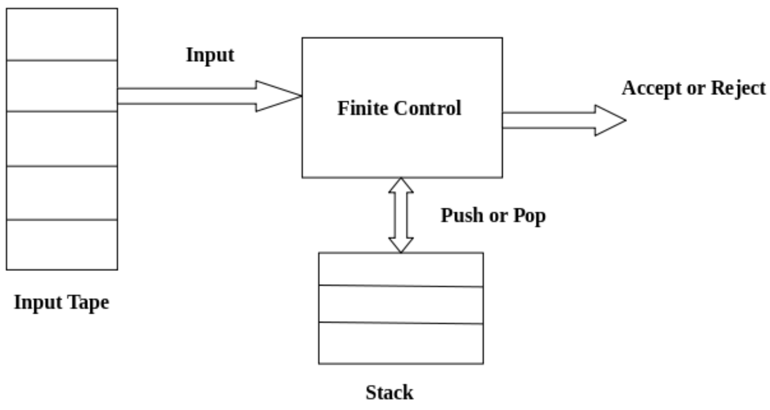


Fig: Pushdown Automata

PDA Components:

Input tape: The input tape is divided in many cells or symbols. The input head is read-only and may only move from left to right, one symbol at a time.

Finite control: The finite control has some pointer which points the current symbol which is to be read.

Stack: The stack is a structure in which we can push and remove the items from one end only. It has an infinite size. In PDA, the stack is used to store the items temporarily.

Formal definition of PDA:

The PDA can be defined as a collection of 7 components:

Q: the finite set of states

∑: the input set

Γ: a stack symbol which can be pushed and popped from the stack

q0: the initial state

Z: a start symbol which is in Γ.

F: a set of final states

δ: mapping function which is used for moving from current state to next state.

Instantaneous Description (ID)

‘

ID is an informal notation of how a PDA computes an input string and make a decision that string is accepted or rejected.

An instantaneous description is a triple (q, w, α) where:

q describes the current state.

w describes the remaining input.

α describes the stack contents, top at the left.

Turnstile Notation:

⊢ sign describes the turnstile notation and represents one move.

⊢* sign describes a sequence of moves.

For example,

(p, b, T) ⊢ (q, w, α)

In the above example, while taking a transition from state p to q, the input symbol 'b' is consumed, and the top of the stack 'T' is represented by a new string α.

Example 1:

Design a PDA for accepting a language $\{a^n b^{2n} \mid n>=1\}$.

Solution: In this language, n number of a's should be followed by 2n number of b's. Hence, we will apply a very simple logic, and that is if we read single 'a', we will push two a's onto the stack. As soon as we read 'b' then for every single 'b' only one 'a' should get popped from the stack.

The ID can be constructed as follows:

1. δ(q0, a, Z) = (q0, aaZ)
2. δ(q0, a, a) = (q0, aaa)

Now when we read b, we will change the state from q0 to q1 and start popping corresponding 'a'. Hence,

1. δ(q0, b, a) = (q1, ε)

‘

Thus this process of popping 'b' will be repeated unless all the symbols are read. Note that popping action occurs in state q1 only.

1. δ(q1, b, a) = (q1, ε)

After reading all b's, all the corresponding a's should get popped. Hence when we read ε as input symbol then there should be nothing in the stack. Hence the move will be:

1. δ(q1, ε, Z) = (q2, ε)

Where

PDA = ({q0, q1, q2}, {a, b}, {a, Z}, δ, q0, Z, {q2})

We can summarize the ID as:

1. δ(q0, a, Z) = (q0, aaZ)
2. δ(q0, a, a) = (q0, aaa)
3. δ(q0, b, a) = (q1, ε)
4. δ(q1, b, a) = (q1, ε)
5. δ(q1, ε, Z) = (q2, ε)

Now we will simulate this PDA for the input string "aaabbbbbb".

1. δ(q0, aaabbbbbb, Z) ⊢ δ(q0, aabbbbbb, aaZ)
2. ⊢ δ(q0, abbbbbb, aaaaZ)
3. ⊢ δ(q0, bbbbbb, aaaaaaZ)
4. ⊢ δ(q1, bbbbb, aaaaaZ)
5. ⊢ δ(q1, bbbb, aaaaZ)
6. ⊢ δ(q1, bbb, aaaZ)
7. ⊢ δ(q1, bb, aaZ)
8. ⊢ δ(q1, b, aZ)
9. ⊢ δ(q1, ε, Z)
10. ⊢ δ(q2, ε)
11. ACCEPT

Example 2:

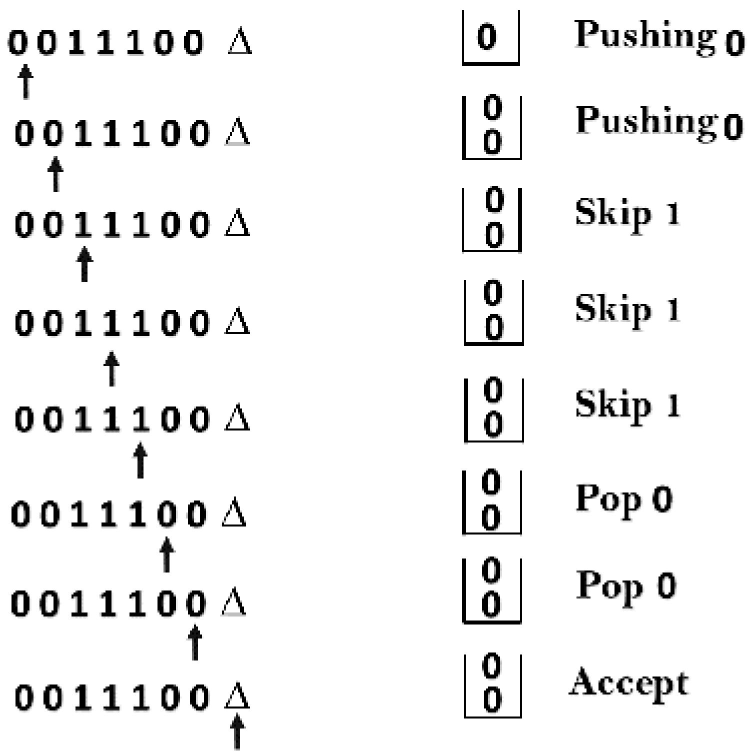
Design a PDA for accepting a language $\{0^n1^m0^n \mid m, n>=1\}$.

‘

Solution: In this PDA, n number of 0's are followed by any number of 1's followed n number of 0's. Hence the logic for design of such PDA will be as follows:

Push all 0's onto the stack on encountering first 0's. Then if we read 1, just do nothing. Then read 0, and on each read of 0, pop one 0 from the stack.

For instance:



This scenario can be written in the ID form as:

1. δ(q0, 0, Z) = δ(q0, 0Z)
2. δ(q0, 0, 0) = δ(q0, 00)
3. δ(q0, 1, 0) = δ(q1, 0)
4. δ(q0, 1, 0) = δ(q1, 0)
5. δ(q1, 0, 0) = δ(q1, ε)
6. δ(q0, ε, Z) = δ(q2, Z) (ACCEPT state)

Now we will simulate this PDA for the input string "0011100".

1. δ(q0, 0011100, Z) ⊢ δ(q0, 011100, 0Z)
2. ⊢ δ(q0, 11100, 00Z)
3. ⊢ δ(q0, 1100, 00Z)
4. ⊢ δ(q1, 100, 00Z)

5. ⊢ δ(q1, 00, 00Z)
6. ⊢ δ(q1, 0, 0Z)
7. ⊢ δ(q1, ε, Z)
8. ⊢ δ(q2, Z)
9. ACCEPT

PDA Acceptance

A language can be accepted by Pushdown automata using two approaches:

1. Acceptance by Final State: The PDA is said to accept its input by the final state if it enters any final state in zero or more moves after reading the entire input.

Let P =(Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by the final state can be defined as:

1. L(PDA) = {w | (q0, w, Z) ⊢* (p, ε, ε), q ∈ F}

2. Acceptance by Empty Stack: On reading the input string from the initial configuration for some PDA, the stack of PDA gets empty.

Let P =(Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by empty stack can be defined as:

1. N(PDA) = {w | (q0, w, Z) ⊢* (p, ε, ε), q ∈ Q}

Equivalence of Acceptance by Final State and Empty Stack

- If L = N(P1) for some PDA P1, then there is a PDA P2 such that L = L(P2). That means the language accepted by empty stack PDA will also be accepted by final state PDA.
- If there is a language L = L (P1) for some PDA P1 then there is a PDA P2 such that L = N(P2). That means language accepted by final state PDA is also acceptable by empty stack PDA.

Example:

‘

Construct a PDA that accepts the language L over {0, 1} by empty stack which accepts all the string of 0's and 1's in which a number of 0's are twice of number of 1's.

Solution:

There are two parts for designing this PDA:

- If 1 comes before any 0's
- If 0 comes before any 1's.

We are going to design the first part i.e. 1 comes before 0's. The logic is that read single 1 and push two 1's onto the stack. Thereafter on reading two 0's, POP two 1's from the stack. The δ can be

1. δ(q0, 1, Z) = (q0, 11, Z) Here Z represents that stack is empty
2. δ(q0, 0, 1) = (q0, ε)

Now, consider the second part i.e. if 0 comes before 1's. The logic is that read first 0, push it onto the stack and change state from q0 to q1. [Note that state q1 indicates that first 0 is read and still second 0 has yet to read].

Being in q1, if 1 is encountered then POP 0. Being in q1, if 0 is read then simply read that second 0 and move ahead. The δ will be:

1. δ(q0, 0, Z) = (q1, 0Z)
2. δ(q1, 0, 0) = (q1, 0)
3. δ(q1, 0, Z) = (q0, ε) (indicate that one 0 and one 1 is already read, so simpl y read the second 0)
4. δ(q1, 1, 0) = (q1, ε)

Now, summarize the complete PDA for given L is:

1. δ(q0, 1, Z) = (q0, 11Z)
2. δ(q0, 0, 1) = (q1, ε)
3. δ(q0, 0, Z) = (q1, 0Z)
4. δ(q1, 0, 0) = (q1, 0)
5. δ(q1, 0, Z) = (q0, ε)
6. δ(q0, ε, Z) = (q0, ε) ACCEPT state

‘

Non-deterministic Pushdown Automata

The non-deterministic pushdown automata is very much similar to NFA. We will discuss some CFGs which accepts NPDA.

The CFG which accepts deterministic PDA accepts non-deterministic PDAs as well. Similarly, there are some CFGs which can be accepted only by NPDA and not by DPDA. Thus NPDA is more powerful than DPDA.

Example:

Design PDA for Palindrome strips.

Solution:

Suppose the language consists of string L = {aba, aa, bb, bab, bbabb, aabaa, ...]. The string can be odd palindrome or even palindrome. The logic for constructing PDA is that we will push a symbol onto the stack till half of the string then we will read each symbol and then perform the pop operation. We will compare to see whether the symbol which is popped is similar to the symbol which is read. Whether we reach to end of the input, we expect the stack to be empty.

This PDA is a non-deterministic PDA because finding the mid for the given string and reading the string from left and matching it with from right (reverse) direction leads to non-deterministic moves. Here is the ID.

‘

1. δ(q1, a, Z) = (q1, aZ)
2. δ(q0, b, Z) = (q1, bZ)
3. δ(q0, a, a) = (q1, aa)
4. δ(q1, a, b) = (q1, ab)
5. δ(q1, a, b) = (q1, ba)
6. δ(q1, b, b) = (q1, bb)

Pushing the symbols onto the stack

7. δ(q1, a, a) = (q2, ε)
8. δ(q1, b, b) = (q2, ε)
9. δ(q2, a, a) = (q2, ε)
10. δ(q2, b, b) = (q2, ε)
11. δ(q2, ε, Z) = (q2, ε)

Popping the symbols on reading the same kind of symbol

Simulation of abaaba

1. δ(q1, abaaba, Z) Apply rule 1
2. ⊢ δ(q1, baaba, aZ) Apply rule 5
3. ⊢ δ(q1, aaba, baZ) Apply rule 4
4. ⊢ δ(q1, aba, abaZ) Apply rule 7
5. ⊢ δ(q2, ba, baZ) Apply rule 8
6. ⊢ δ(q2, a, aZ) Apply rule 7
7. ⊢ δ(q2, ε, Z) Apply rule 11
8. ⊢ δ(q2, ε) Accept

CFG to PDA Conversion

The first symbol on R.H.S. production must be a terminal symbol. The following steps are used to obtain PDA from CFG is:

Step 1: Convert the given productions of CFG into GNF.

Step 2: The PDA will only have one state {q}.

Step 3: The initial symbol of CFG will be the initial symbol in the PDA.

Step 4: For non-terminal symbol, add the following rule:

1. δ(q, ε, A) = (q, α)

Where the production rule is A → α

Step 5: For each terminal symbols, add the following rule:

1. δ(q, a, a) = (q, ε) **for** every terminal symbol

Example 1:

Convert the following grammar to a PDA that accepts the same language.

1. S → 0S1 | A
2. A → 1A0 | S | ε

Solution:

The CFG can be first simplified by eliminating unit productions:

1. S → 0S1 | 1S0 | ε

Now we will convert this CFG to GNF:

1. S → 0SX | 1SY | ε
2. X → 1
3. Y → 0

The PDA can be:

R1: `δ(q, ε, S) = {(q, 0SX) | (q, 1SY) | (q, ε)}`
R2: `δ(q, ε, X) = {(q, 1)}`
R3: `δ(q, ε, Y) = {(q, 0)}`
R4: `δ(q, 0, 0) = {(q, ε)}`
R5: `δ(q, 1, 1) = {(q, ε)}`

Example 2:

'

Construct PDA for the given CFG, and test whether 0104 is acceptable by this PDA.

1. S → 0BB
2. B → 0S | 1S | 0

Solution:

The PDA can be given as:

1. A = {(q), (0, 1), (S, B, 0, 1), δ, q, S, ?}

The production rule δ can be:

R1: `δ(q, ε, S) = {(q, 0BB)}`
R2: `δ(q, ε, B) = {(q, 0S) | (q, 1S) | (q, 0)}`
R3: `δ(q, 0, 0) = {(q, ε)}`
R4: `δ(q, 1, 1) = {(q, ε)}`

Testing 010^4 i.e. 010000 against PDA:

1. δ(q, 010000, S) ⊢ δ(q, 010000, 0BB)
2. ⊢ δ(q, 10000, BB) R1
3. ⊢ δ(q, 10000,1SB) R3
4. ⊢ δ(q, 0000, SB) R2
5. ⊢ δ(q, 0000, 0BBB) R1
6. ⊢ δ(q, 000, BBB) R3
7. ⊢ δ(q, 000, 0BB) R2
8. ⊢ δ(q, 00, BB) R3
9. ⊢ δ(q, 00, 0B) R2
10. ⊢ δ(q, 0, B) R3
11. ⊢ δ(q, 0, 0) R2
12. ⊢ δ(q, ε) R3
13. ACCEPT

Thus 010^4 is accepted by the PDA.

Example 3:

Draw a PDA for the CFG given below:

1. S → aSb

‘

2. S → a | b | ε

Solution:

The PDA can be given as:

1. P = {(q), (a, b), (S, a, b, z0), δ, q, z0, q}

The mapping function δ will be:

R1: `δ(q, ε, S) = {(q, aSb)}`
R2: `δ(q, ε, S) = {(q, a) | (q, b) | (q, ε)}`
R3: `δ(q, a, a) = {(q, ε)}`
R4: `δ(q, b, b) = {(q, ε)}`
R5: `δ(q, ε, z0) = {(q, ε)}`

Simulation: Consider the string aaabb

1. δ(q, εaaabb, S) ⊢ δ(q, aaabb, aSb) R3
2. ⊢ δ(q, εaabb, Sb) R1
3. ⊢ δ(q, aabb, aSbb) R3
4. ⊢ δ(q, εabb, Sbb) R2
5. ⊢ δ(q, abb, abb) R3
6. ⊢ δ(q, bb, bb) R4
7. ⊢ δ(q, b, b) R4
8. ⊢ δ(q, ε, z0) R5
9. ⊢ δ(q, ε)
10. ACCEPT

‘

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 ∈ L**.

Applications of Pumping Lemma

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

Problem

Find out whether the language **L = {$x^ny^nz^n$ | n ≥ 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.

CFL Closure Properties •

Theorem#1: The context-free languages are closed under concatenation, union, and Kleene closure. •

Proof: Start with 2 CFL L(H1) and L(H2) generated by H1 = (N1,T1,R1,s1) and H2 = (N2,T2,R2,s2). Assume that the alphabets and rules are disjoint.

Concatenation:

Formed by L(H1)·L(H2) or a string in L(H1) followed by a string in L(H2) which can be generated by L(H3) generated by H3 = (N3,T3,R3,s3). N3 = N1 ∪ N2, T3

‘

= T1 ∪ T2, R3 = R1 ∪ R2 ∪ {s3 -->s1s2} where s3 ◊s1s2 is a new rule introduced. The new rule generates a string of L(H1) then a string of L(H2). Then L(H1) ·L(H2) is context-free.

Union:

Formed by L(H1) ∪ L(H2) or a string in L(H1) or a string in L(H2). It is generated by L(H3) generated by H4 = (N4,T4,R4,s4) where N4 = N1 ∪ N2, T4 = T1 ∪ T2, and R4 = R1 ∪ R2 ∪ {s4-->s1, s4 ◊ s2}, the new rules added will create a string of L(H1) or L(H2). Then L(H1) ∪ L(H2) is context-free.

Kleene:

Formed by L(H1)* is generated by the grammar L(H5) generated by H5 = (N1,T1,R5,s1) with R5 = R1 ∪ {s1◊e, s1◊s1s1}. L(H5) includes e, every string in L(H1), and through i-1 applications of s1◊s1s1, every string in L(H1)i. Then L(H1)* is generated by H5 and is context-free.

Closure Properties of Context Free Languages

- Difficulty Level : Medium
- Last Updated : 28 Jun, 2021

Context Free Languages (CFLs) are accepted by pushdown automata. Context free languages can be generated by context free grammars, which have productions (substitution rules) of the form :
A -> ρ (where A ∈ N and ρ ∈ (T ∪ N)* and N is a non-terminal and T is a terminal)

Properties of Context Free Languages
Union : If L1 and L2 are two context free languages, their union L1 ∪ L2 will also be context free. For example,
L1 = { $a^nb^nc^m$ | m >= 0 and n >= 0 } and L2 = { $a^nb^mc^m$ | n >= 0 and m >= 0 }
L3 = L1 ∪ L2 = { $a^nb^nc^m$ ∪ $a^nb^mc^m$ | n >= 0, m >= 0 } is also context free.
L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their union says either of two conditions to be true. So it is also context free language.
Note: So CFL are closed under Union.

Concatenation : If L1 and If L2 are two context free languages, their concatenation L1.L2 will also be context free. For example,
L1 = { a^nb^n | n >= 0 } and L2 = { c^md^m | m >= 0 }
L3 = L1.L2 = { $a^nb^nc^md^m$ | m >= 0 and n >= 0} is also context free.
L1 says number of a's should be equal to number of b's and L2 says number of c's should be equal to number of d's. Their concatenation says first number of a's should be equal to number of b's, then number of c's should be equal to number

‘

of d’s. So, we can create a PDA which will first push for a’s, pop for b’s, push for c’s then pop for d’s. So it can be accepted by pushdown automata, hence context free.
Note: So CFL are closed under Concatenation.

Kleene Closure : If L1 is context free, its Kleene closure L1* will also be context free. For example,
L1 = { $a^n b^n$ | n >= 0 }
L1* = { $a^n b^n$ | n >= 0 }* is also context free.
Note : So CFL are closed under Kleen Closure.

Intersection and complementation : If L1 and If L2 are two context free languages, their intersection L1 ∩ L2 need not be context free. For example,
L1 = { $a^n b^n c^m$ | n >= 0 and m >= 0 } and L2 = ($a^m b^n c^n$ | n >= 0 and m >= 0 }
L3 = L1 ∩ L2 = { $a^n b^n c^n$ | n >= 0 } need not be context free.
L1 says number of a’s should be equal to number of b’s and L2 says number of b’s should be equal to number of c’s. Their intersection says both conditions need to be true, but push down automata can compare only two. So it cannot be accepted by pushdown automata, hence not context free.
Similarly, complementation of context free language L1 which is ∑* – L1, need not be context free.

Deterministic PDAs and DCFLs • Definition: A Deterministic Pushdown Automaton (DPDA) is a 7-tuple, M = (Q, Σ, Γ, δ, q0, Z0, A), where Q = finite set of states, Σ = input alphabet, Γ = stack alphabet, q0 ∈ Q = the initial state, Z0 ∈Γ = bottom of stack marker (or initial stack symbol), and δ: Q × (Σ ∪{L}) × Γ → Q × Γ* = the transition function (not necessarily total). Specifically, [1] if d(q, a, Z) is defined for some a ∈Σ and Z ∈Γ, then d(q, L, Z) = Φ and |d(q, a, Z)|= 1. 103 [2] Conversely, if d(q, L, Z) ≠ Φ, for some Z, then d(q, a, Z) = Φ, for all a ∈Σ, and |d(q, L, Z)|= 1. • NOTE: DPDAs can accept their input either by final state or by empty stack – just as for the non-deterministic model. We therefore define Dstk and Dste, respectively, as the corresponding families of Deterministic Context-free Languages accepted by a DPDA by empty stack and final state.

‘

6.3.2 Deterministic PDA (DPDA)

Deterministic PDA (DPDA) is just like DFA, which has *at most one choice* to move for certain input. A PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is deterministic if it satisfies both the conditions given as follows:

1. For any $q \in Q$, $a \in (\Sigma \cup \{\epsilon\})$, and $Z \in \Gamma$, $\delta(q, a, Z)$ has at most one choice of move.
2. For any $q \in Q$, and $Z \in \Gamma$, if $\delta(q, \epsilon, Z)$ is defined i.e. $\delta(q, \epsilon, Z) \neq \phi$, then $\delta(q, a, Z) = \phi$ for all $a \in \Sigma$

Example : Consider a DPDA $M = (\{q_0, q_1\}, \{a, c\}, \{a, Z_0\}, \delta, q_0, Z_0, \phi)$ accepting the language $\{a^n c a^n : n \geq 1\}$, where δ is defined as follows:

$$\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}$$
$$\delta(q_0, a, a) = \{(q_0, aa)\},$$
$$\delta(q_0, c, a) = \{(q_1, a)\},$$
$$\delta(q_1, a, a) = \{(q_1, \epsilon)\}, \text{ and } \delta(q_1, \epsilon, Z_0) = \{(q_1, \epsilon)\}$$

Check whether the string $w = aacaa$ is accepted by empty stack or not ?

Solution :

We see that in each transition DPDA has at most one move. Initial configuration is $(q_0, aacaa, Z_0)$. Following are the possible moves.

$$(q_0, aacaa, Z_0) \to (q_0, acaa, aZ_0) \to (q_0, caa, aaZ_0) \to (q_1, aa, aaZ_0)$$
$$\downarrow$$
$$(q_1, \epsilon, \epsilon) \leftarrow (q_1, \epsilon, Z_0) \leftarrow (q_1, a, aZ_0)$$

Hence, the string $w = aacaa$ is accepted by empty stack.

As we have discussed in earlier chapters that DFA and NFA are equivalent with respect to the language acceptance, but the same is not true for the PDA.

For example, language $L = \{ww^R : w \in (a \cup b)^*\}$ is accepted by nondeterministic PDA, can not by any deterministic PDA. A nondeterministic PDA can not be converted into equivalent deterministic PDA, but all DCFLs which are accepted by DPDA, are also accepted by NPDA. So, we say that deterministic PDA is a proper subset of nondeterministic PDA. Hence, the power of nondeterministic PDA is more as compared to deterministic PDA.

Context-sensitive Grammar (CSG) and Language (CSL)

Context-Sensitive Grammar –

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

$$\alpha \to \beta$$

where **$\alpha, \beta \in (V \cup T)^+$** and **$|\alpha| \leq |\beta|$**

Where α and β are strings of non-terminals and terminals.

Context-sensitive grammars are **more powerful** than context-free grammars because there are some languages that can be described by CSG but not by context-free grammars and CSL are less powerful than Unrestricted grammar.

That’s why context-sensitive grammars are positioned between context-free and unrestricted grammars in the Chomsky hierarchy.

Context-sensitive grammar has 4-tuples. **G = {N, Σ, P, S}**, Where
N = Set of non-terminal symbols
Σ = Set of terminal symbols
S = Start symbol of the production
P = Finite set of productions
All rules in P are of the form $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$

Context-sensitive Language: The language that can be defined by context-sensitive grammar is called CSL. Properties of CSL are :

- Union, intersection and concatenation of two context-sensitive languages is context-sensitive.
- Complement of a context-sensitive language is context-sensitive.

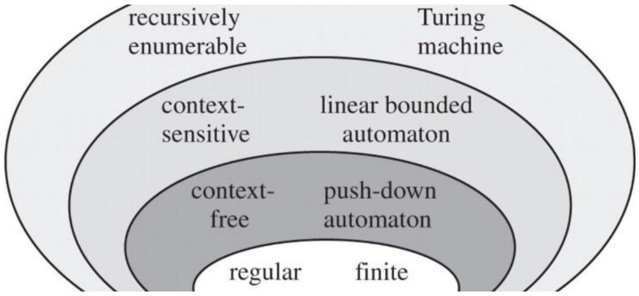
Example –

Consider the following CSG.
S → abc/aAbc
Ab → bA
Ac → Bbcc
bB → Bb
aB → aa/aaA
What is the language generated by this grammar?
Solution:
S → aAbc
→ abAc
→ abBbcc
→ aBbbcc
→ aaAbbcc
→ aabAbcc
→ aabbAcc
→ aabbBbccc
→ aabBbbccc
→ aaBbbbccc

‘

→ aaabbbccc
The language generated by this grammar is {$\mathbf{a^n b^n c^n}$ | n≥1}.

Context Sensitive Grammar And Linear Bounded Automata



A context sensitive grammar (CSG) is a grammar where all productions are of the form **αAβ → αγβ** where **γ ≠ ε**.
Advertisements

During derivation non-terminal ***A*** will be changed to **γ** only when it is present in the context of **α** and **β.**

Note the constraint that the replacement string **γ ≠ ε ; as a consequence we have **α** □ **β** implies **|α| ≤ |β|***

CSG is a Noncontracting grammar.

Formal Definition of Context Sensitive Grammar

A context sensitive grammar G = (N, Σ, P, S), where

- N is a set of non-terminal symbols
- Σ is a set of terminal symbols
- S is the start symbol, and
- P is a set of production rules, of the form α***A***β → α**γ**β , where A in N, α, β ∈ (N ∪ Σ) and γ ∈ $(N \cup \Sigma)^+$

The production S → ε is also allowed if S is the start symbol and it does not appear on the right side of any production.

Linear Bounded Automata

‘

Linear Bounded Automata (LBA) is a single tape Turing Machine with two special tape symbols call them left marker **<** and the right marker **>.**
The transitions should satisfy these conditions:

- It should not replace the marker symbols by any other symbol.
- It should not write on cells beyond the marker symbols.

Thus the initial configuration will be : < q0a1a2a3a4a5…..an >

Formal Definition

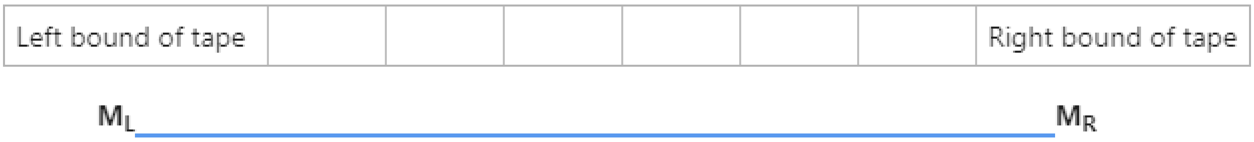
Formally Linear Bounded Automata is a non-deterministic Turing Machine , M = (Q, Σ, Γ, δ, ε, q0, <, >, t, r)

- Q is set of all states
- Σ is set of all terminals
- Γ is set of all tape symbols, Σ ⊂ Γ
- δ is set of transitions
- ε is blank symbols or null
- < is left marker and > is right marker
- t is accept state
- r is reject state

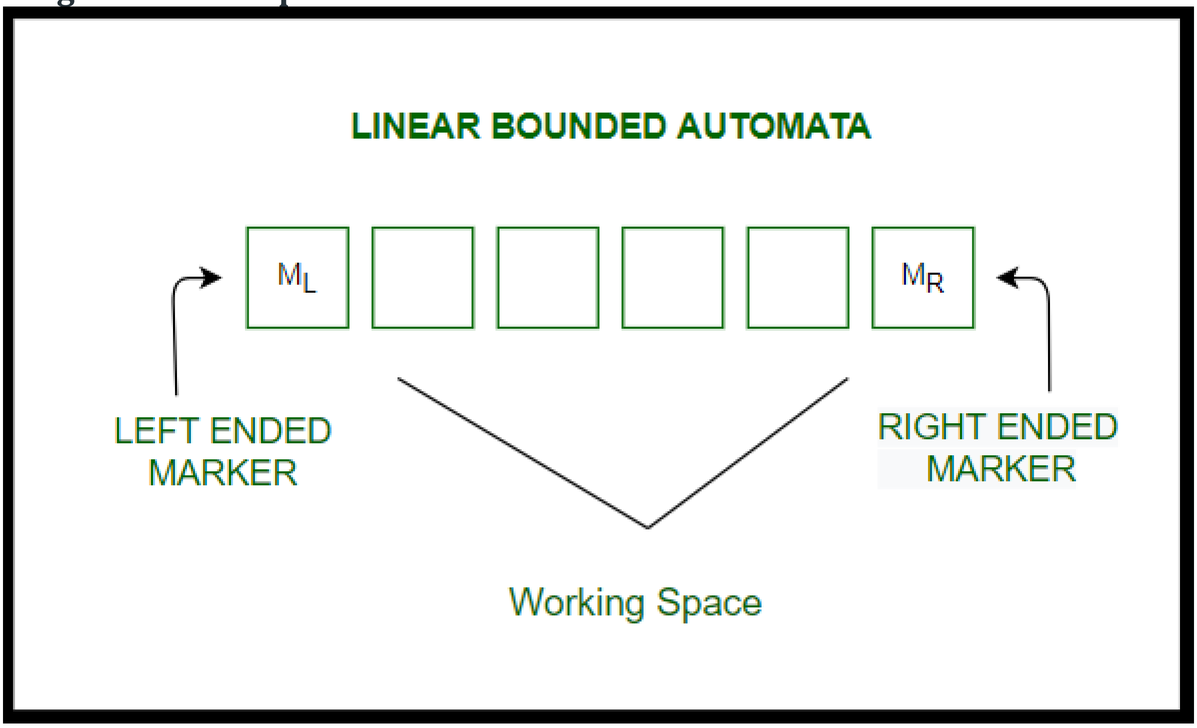
Introduction to Linear Bounded Automata (LBA)

History :
In 1960, associate degree automaton model was introduced by **Myhill** and these days this automation model is understood as deterministic linear bounded automaton. After this, another scientist named **Landweber** worked on this and proposed that the languages accepted by a deterministic LBA are continually context-sensitive languages.
In 1964, **Kuroda** introduced a replacement and a lot of general models specially for non-deterministic linear bounded automata, and established that the languages accepted by the non-deterministic linear bounded automata are exactly the context-sensitive languages.
Introduction to Linear Bounded Automata :
A Linear Bounded Automaton (LBA) is similar to Turing Machine with some properties stated below:

- Turing Machine with Non-deterministic logic,
- Turing Machine with Multi-track, and
- Turing Machine with a bounded finite length of the tape.



Tuples Used in LBA :
LBA can be defined with eight tuples (elements that help to design automata) as:
```
M = (Q , T , E , q0 , ML , MR , S , F),

where,
Q -> A finite set of transition states
T -> Tape alphabet
E -> Input alphabet
q0 -> Initial state
ML -> Left bound of tape
MR -> Right bound of tape
S -> Transition Function
F -> A finite set of final states
```
Diagrammatic Representation of LBA :



Examples:
Languages that form LBA with tape as shown above,= {$a^{n!}$ | n >= 0}
- L = {wn | w from {a, b}+, n >= 1}
- L = {wwwR | w from {a, b}+}

Facts :

Suppose that a given LBA M has

--> q states,

--> m characters within the tape alphabet, and

--> the input length is n

1. Then M can be in at most f(n) = q * n * mn configurations i.e. a tape of n cells and m symbols, we are able to have solely mn totally different tapes.
2. The tape head is typically on any of the n cells which we have a tendency to are typically death penalty in any of the q states.

K

‘