Theory of Computing
Lec-5

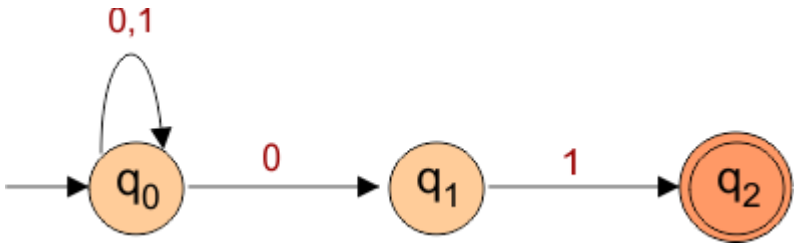
NFA

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

- Language for given question is given below

L= {01, 0101, 0000101, 101, 101001……..}

NFA for above language is



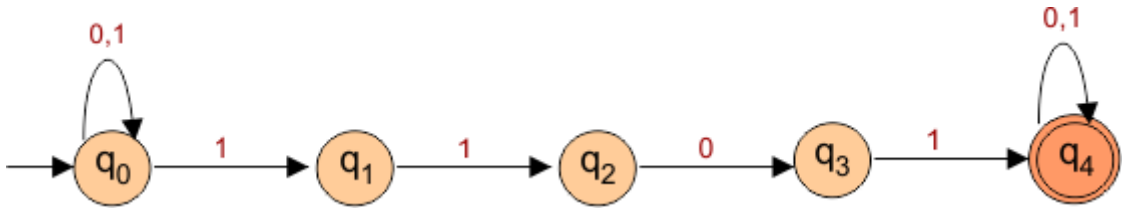
NFA

- Construct an NFA in which all the string contains a substring 1101

- Language for given question is given below

  L= {1101, 110101, 1101000101, 01101, 1011011……..}

  NFA for above language is



Equivalence of DFA & NFA

We are going to prove that the DFA obtained from NFA by the conversion algorithm accepts the same language as the NFA.
NFA that recognizes a language L is denoted by $M_1 = < Q_1 , \Sigma , q_{1,0} , \delta_1 , A_1 >$ and DFA obtained by the conversion is denoted by $M_2 = < Q_2, \Sigma , q_{2,0} , \delta_2 , A_2 >$

First we are going to prove by induction on strings that $\delta_1^*(q_{1,0} , w) = \delta_2^*(q_{2,0} , w)$ for any string w. When it is proven, it obviously implies that NFA M_1 and DFA M_2 accept the same strings.

Theorem: For any string w, $\delta_1^*(q_{1,0} , w) = \delta_2^*(q_{2,0} , w)$.

Proof: This is going to be proven by induction on w.

$q_{1,0}=q_{2,0}$ are the initial states of nfa and dfa

Basis Step: For $w = \Lambda$,

$\delta_2^*(q_{2,0} , \Lambda) = q_{2,0}$ by the definition of δ_2^*.
$= \{ q_{1,0} \}$ by the construction of DFA M_2.
$= \delta_1^*(q_{1,0} , \Lambda)$ by the definition of δ_1^*. |

$\delta^*(q, \wedge) = q$
$\delta^*(q, wa) = \delta(\delta^*(q, w) , a)$

Inductive Step: Assume that $\delta_1^*(q_{1,0} , w) = \delta_2^*(q_{2,0} , w)$ for an arbitrary string w. --- Induction Hypothesis

For the string w and an arbitrry symbol a in Σ ,
$\delta_1^*(q_{1,0} , wa) = \cup_{p \in \delta_1^*(q_{1,0},w)} \delta_1(p, a)$

$= \delta_2(\delta_1^*(q_{1,0} , w) , a)$
$= \delta_2(\delta_2^*(q_{2,0} , w) , a)$
$= \delta_2^*(q_{2,0} , wa)$

Thus for any string w $\delta_1^*(q_{1,0} , w) = \delta_2^*(q_{2,0} , w)$ holds.

Conversion from NFA to DFA

The following steps are followed to convert a given NFA to a DFA-

Step-01:

- Let Q' be a new set of states of the DFA. Q' is null in the starting.
- Let T' be a new transition table of the DFA.

Step-02:

- Add start state of the NFA to Q'.
- Add transitions of the start state to the transition table T'.
- If start state makes transition to multiple states for some input alphabet, then treat those multiple states as a single state in the DFA.

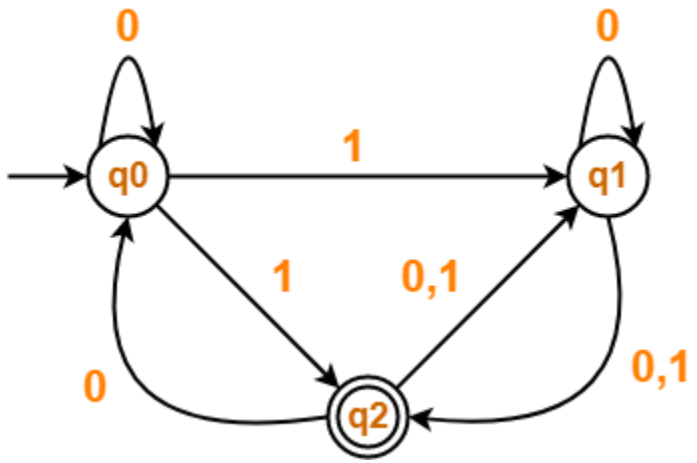
Step-03:

- If any new state is present in the transition table T',
- Add the new state in Q'.
- Add transitions of that state in the transition table T'.

Step-04:

- Keep repeating Step-03 until no new state is present in the transition table T'.
- Finally, the transition table T' so obtained is the complete transition table of the required DFA.

Conversion from NFA to DFA



Transition table for the given Non-Deterministic Finite Automata (NFA) is-

State / Alphabet	0	1
→q0	q0	q1, *q2
q1	q1, *q2	*q2
*q2	q0, q1	q1

Conversion from NFA to DFA

Step-01:

- Let Q' be a new set of states of the Deterministic Finite Automata (DFA).
- Let T' be a new transition table of the DFA.

State / Alphabet	0	1
→q0	q0	q1, *q2
q1	q1, *q2	*q2
*q2	q0, q1	q1

Step-02:

- Add transitions of start state q0 to the transition table T'.

State / Alphabet	0	1
→q0	{q0}	{q1, q2}

Step-03:

- New state present in state Q' is {q1,q2}.
- Add transitions for set of states {q1, q2} to the transition table T'.

State / Alphabet	0	1
→q0	{q0}	{q1, q2}
{q1, q2}	{q0, q1, q2}	{q1, q2}

Conversion from NFA to DFA

Step-04:

- New state present in state Q' is {q0, q1, q2}.
- Add transitions for set of states {q0, q1, q2} to the transition table T'.

State / Alphabet	0	1
→q0	q0	q1, *q2
q1	q1, *q2	*q2
*q2	q0, q1	q1

State / Alphabet	0	1
→q0	{q0}	{q1, q2}
{q1, q2}	{q0, q1, q2}	{q1, q2}
{q0, q1, q2}	{q0, q1, q2}	{q1, q2}

Step-05:

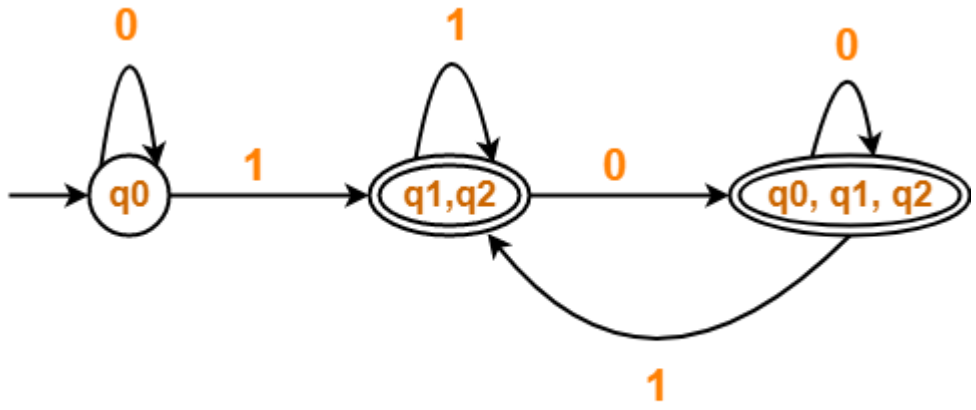
- Since no new states are left to be added in the transition table T', so we stop.
- States containing q2 as its component are treated as final states of the DFA.

Conversion from NFA to DFA

Finally, Transition table for Deterministic Finite Automata (DFA) is-

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

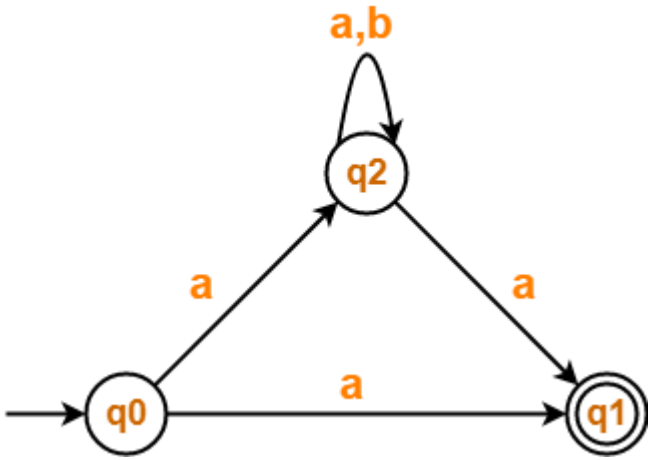
Now, Deterministic Finite Automata (DFA) may be drawn as-



Deterministic Finite Automata (DFA)

Practice Session

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



Practice Session Solution

- Transition table for the given Non-Deterministic Finite Automata (NFA) is-

State / Alphabet	a	b
→q0	{*q1, q2}	–
*q1	–	–
q2	{*q1, q2}	{q2}

Step-01:

- Let Q' be a new set of states of the Deterministic Finite Automata (DFA).
- Let T' be a new transition table of the DFA.

Step-02:

Add transitions of start state q0 to the transition table T'.

State / Alphabet	a	b
→q0	{q1, q2}	Ø (Dead State)

Practice Session Solution

Step-03:

- New state present in state Q' is {q1, q2}.
- Add transitions for set of states {q1, q2} to the transition table T'.

State / Alphabet	a	b
→q0	{q1, q2}	Ø
{q1, q2}	{q1, q2}	{q2}

State / Alphabet	a	b
→q0	{*q1, q2}	–
*q1	–	–
q2	{*q1, q2}	{q2}

Step-04:

- New state present in state Q' is q2.
- Add transitions for state q2 to the transition table T'.

State / Alphabet	a	b
→q0	{q1, q2}	Ø
{q1, q2}	{q1, q2}	{q2}
q2	{q1, q2}	{q2}

Practice Session Solution

Step-05:

- Add transitions for dead state {Ø} to the transition table T.

State / Alphabet	a	b
→q0	{q1, q2}	Ø
{q1, q2}	{q1, q2}	{q2}
q2	{q1, q2}	{q2}
ø	Ø	Ø

State / Alphabet	a	b
→q0	{*q1, q2}	–
*q1	–	–
q2	{*q1, q2}	{q2}

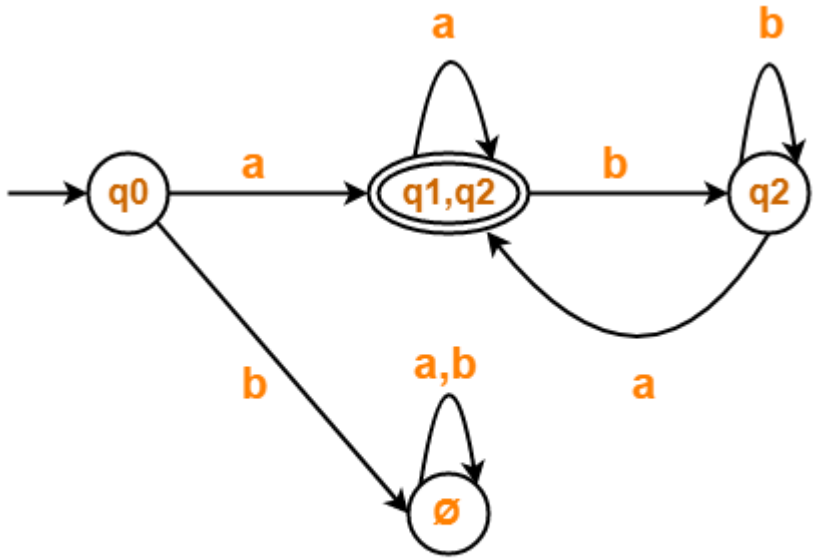
Practice Session Solution

Step-06:

- Since no new states are left to be added in the transition table T', so we stop.
- States containing q1 as its component are treated as final states of the DFA.
- Finally, Transition table for Deterministic Finite Automata (DFA) is-

State / Alphabet	a	b
→q0	{q1, q2}	Ø
{q1, q2}	{q1, q2}	{q2}
q2	{q1, q2}	{q2}
Ø	Ø	Ø

Now, Deterministic Finite Automata (DFA) may be drawn as-

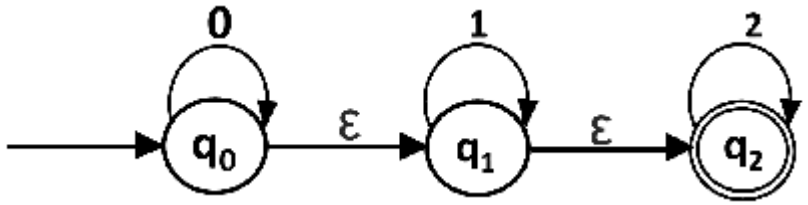


Deterministic Finite Automata (DFA)

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.

Converting NFA with ε to DFA



Transition Diagram of the NFA

States	0	1	2
q0	{q0}	{}	{}
q1	{}	{q1}	{}
q2	{}	{}	{q2}

Converting NFA with ε to DFA

- Step-1: 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}

States	0	1	2
q0	{q0}	{}	{}
q1	{}	{q1}	{}
*q2	{}	{}	*{q2}

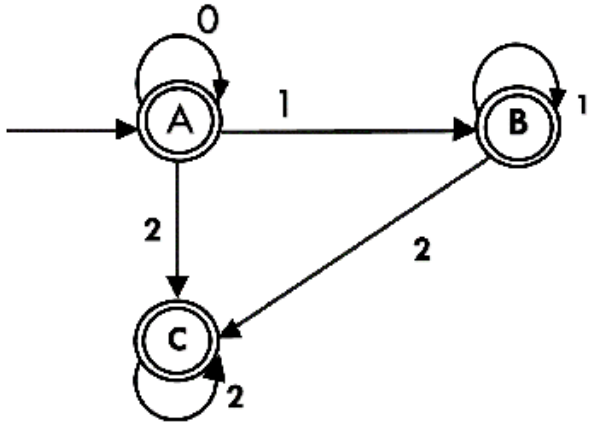
Now we will obtain δ' transition. Let ε-closure(q0) = {q0, q1, q2} call it as **state A**.
{q1,q2} call it as **state B**, {q2} as **state C**

States	0	1	2
{q0,q1,q2}	{q0,q1,q2}	{q1,q2}	{q2}
{q1,q2}	{}	{q1,q2}	{q2}
{q2}	{}	{}	{q2}

Converting NFA with ε to DFA

States	0	1	2
{q0,q1,q2} [A]	{q0,q1,q2} [A]	{q1,q2} [B]	{q2} [C]
{q1,q2} [B]	{}	{q1,q2} [B]	{q2} [C]
{q2} [C]	{}	{}	{q2} [C]

Final DFA will be



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.

End

Theory of Computing
Lec-6

Context Free Grammar (CFG)

A context Free Grammar (CFG) is a 4-tuple such that-

$$\mathbf{G = (V , T , P , S)}$$

where-

V = Finite non-empty set of variables / non-terminal symbols

T = Finite set of terminal symbols

P = Finite non-empty set of production rules of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$

S = Start symbol

Context Free Grammar (CFG)

Example:

S → aSbS

S → bSaS

S → ∈

Consider a grammar G = (V , T , P , S) where-

- V = { S }
- T = { a , b }
- P = { S → aSbS , S → bSaS , S → ∈ }
- S = { S }

Exercise

Answer each part for the following context-free grammar G.

$R \rightarrow XRX \mid S$
$S \rightarrow aT\,b \mid bT\,a$
$T \rightarrow XT\,X \mid X \mid \varepsilon$
$X \rightarrow a \mid b$

a. What are the variables of G?
b. What are the terminals of G?
c. Which is the start variable of G?

Context Free Grammar (CFG)

Ex-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

  r.e. = a*

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

- Production rule for the Regular expression is as follows:

1. A → aA
2. A → ε

Context Free Grammar (CFG)

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

A

aA

aaA rule 1

aaaA rule 1

aaaaA rule 1

aaaaaA rule 1

aaaaaaA rule 1

aaaaaaε rule 2

aaaaaa

Rules

1.A → aA

2.A → ε

Context Free Grammar (CFG)

- Construct a CFG for the regular expression (a+b)*

Solution:

The CFG can be given by,

Production rule (P):

1. S → aS | bS
2. S → ε

(a+b)*={ε , aa,bb,aab……}

For implementing ‘aab’

S

aS

aaS

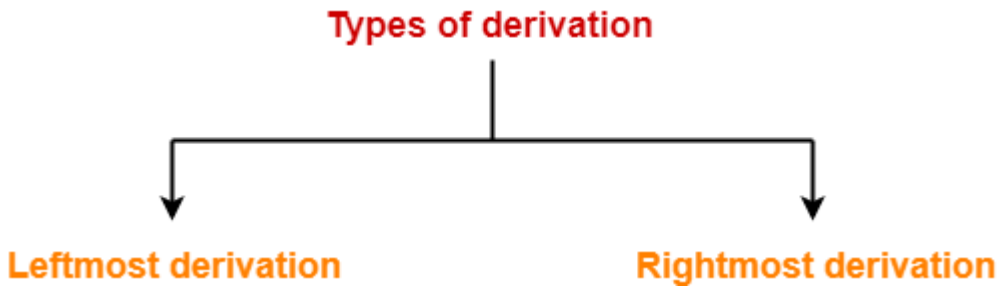
aabS

aab ε

aab

Derivations/ Parse Tree

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



1. Leftmost Derivation-

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

Derivations/ Parse Tree

Example

- Consider the following grammar-

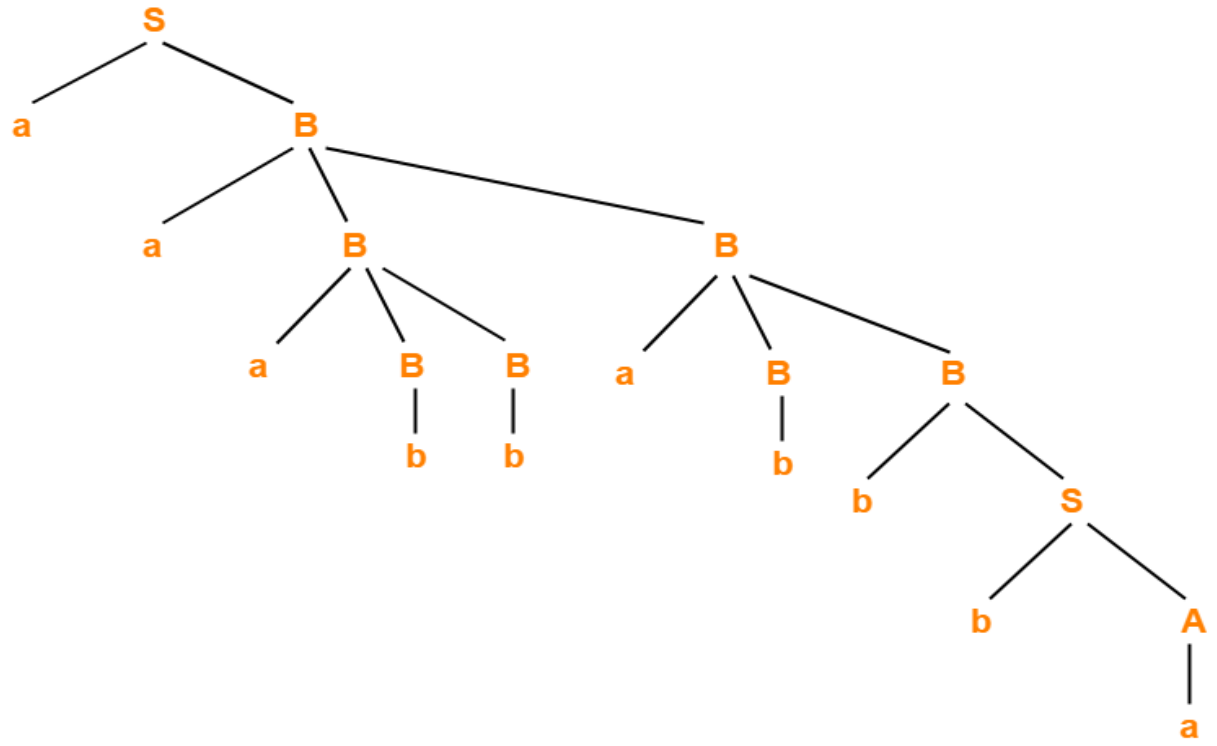
S → aB / bA
S → aS / bAA / a
B → bS / aBB / b

- Let us consider a string w = aaabbabbba
- Now, let us derive the string w using leftmost derivation.

Derivations/ Parse Tree

- Leftmost Derivation for w = aaabbabbba

S → aB / bA
S → aS / bAA /
a
B → bS / aBB /
b
A→ a



Leftmost Derivation Tree

Derivations/ Parse Tree

2. Rightmost Derivation-

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

Consider the following grammar-

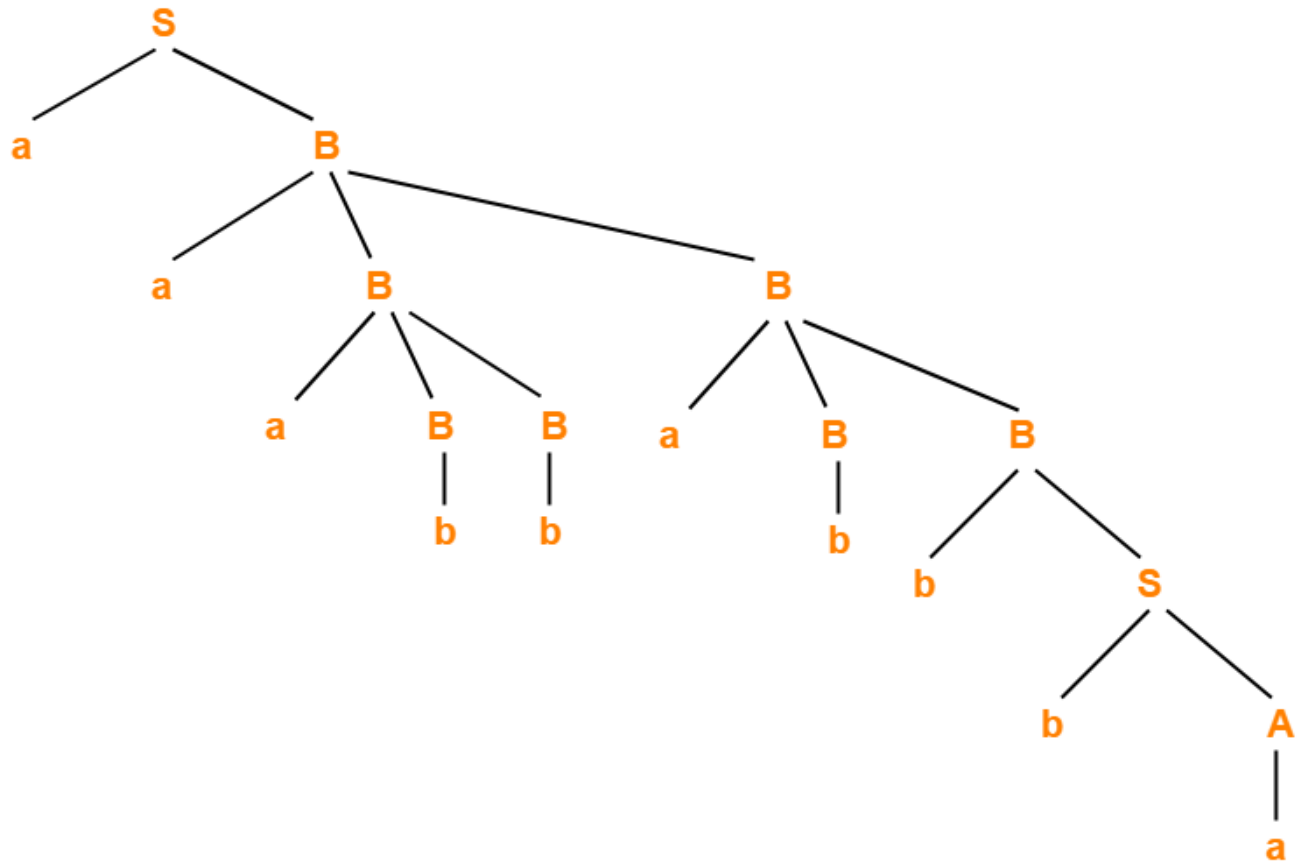
$S \rightarrow aB / bA$
$S \rightarrow aS / bAA / a$
$B \rightarrow bS / aBB / b$
$A \rightarrow a$

Let us consider a string w = aaabbabbba
Now, let us derive the string w using rightmost derivation.

Derivations/ Parse Tree

w = aaabbabbba.

S → aB / bA
S → aS / bAA / a
B → bS / aBB / b
A→ a



Rightmost Derivation Tree

Derivations/ Parse Tree

Some Important Points

For unambiguous grammars, Leftmost derivation and Rightmost derivation represents the same parse tree
For ambiguous grammars, Leftmost derivation and Rightmost derivation represents different parse trees.

In our previous example for w = aaabbabbba

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

Leftmost Derivation Tree = Rightmost Derivation Tree

Problem-1

Consider the grammar-

$$S \rightarrow bB / aA$$

$$A \rightarrow b / bS / aAA$$

$$B \rightarrow a / aS / bBB$$

For the string w = bbaababa, find-

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

Problem-1

w = bbaababa

$S \rightarrow bB / aA$

$A \rightarrow b / bS / aAA$

$B \rightarrow a / aS / bBB$

1. Leftmost Derivation-

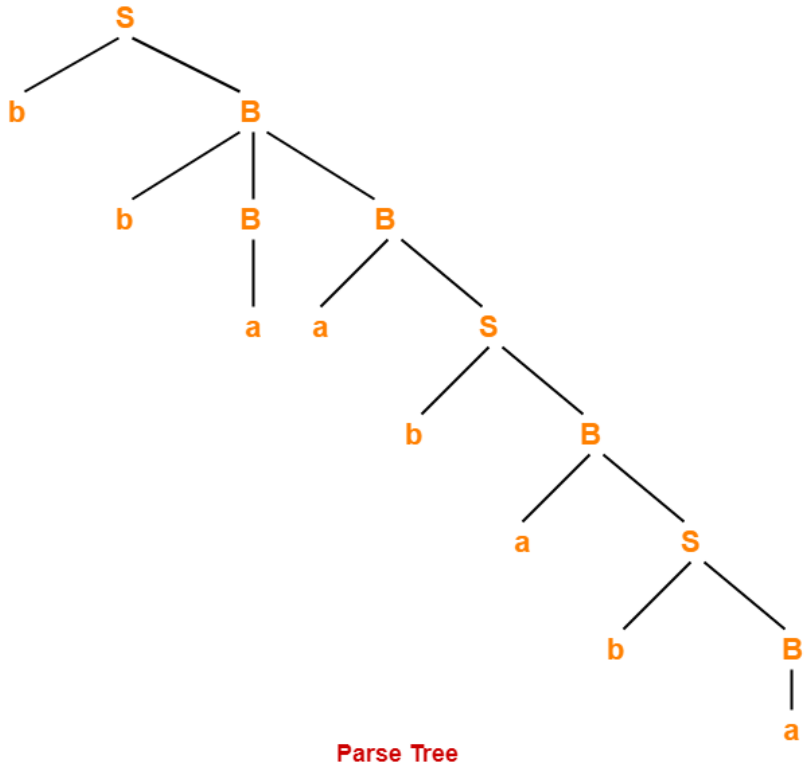
$S \rightarrow b\mathbf{B}$
$\rightarrow bb\mathbf{B}B$ (Using $B \rightarrow bBB$)
$\rightarrow bba\mathbf{B}$ (Using $B \rightarrow a$)
$\rightarrow bbaa\mathbf{S}$ (Using $B \rightarrow aS$)
$\rightarrow bbaab\mathbf{B}$ (Using $S \rightarrow bB$)
$\rightarrow bbaaba\mathbf{S}$ (Using $B \rightarrow aS$)
$\rightarrow bbaabab\mathbf{B}$ (Using $S \rightarrow bB$)
$\rightarrow bbaababa$ (Using $B \rightarrow a$)

2. Rightmost Derivation-

$S \rightarrow b\mathbf{B}$
$\rightarrow bbB\mathbf{B}$ (Using $B \rightarrow bBB$)
$\rightarrow bbBa\mathbf{S}$ (Using $B \rightarrow aS$)
$\rightarrow bbBab\mathbf{B}$ (Using $S \rightarrow bB$)
$\rightarrow bbBaba\mathbf{S}$ (Using $B \rightarrow aS$)
$\rightarrow bbBabab\mathbf{B}$ (Using $S \rightarrow bB$)
$\rightarrow bb\mathbf{B}ababa$ (Using $B \rightarrow a$)
$\rightarrow bbaababa$ (Using $B \rightarrow a$)

Problem-1

- **3. Parse Tree-**



Parse Tree

Whether we consider the leftmost derivation or rightmost derivation, we get the above parse tree.
The reason is given grammar is unambiguous.

Practice Session

- Consider the grammar-

  - $S \rightarrow A1B$
  - $A \rightarrow 0A / \in$
  - $B \rightarrow 0B / 1B / \in$

For the string w = 00101, find-

1. Leftmost derivation

2. Rightmost derivation

3. Parse Tree

Practice Session Solution

$S \rightarrow A1B$

$A \rightarrow 0A / \in$

$B \rightarrow 0B / 1B / \in$

1. Leftmost Derivation-

$S \rightarrow \mathbf{A}1B$
$\rightarrow 0\mathbf{A}1B$ (Using $A \rightarrow 0A$)
$\rightarrow 00\mathbf{A}1B$ (Using $A \rightarrow 0A$)
$\rightarrow 001\mathbf{B}$ (Using $A \rightarrow \in$)
$\rightarrow 0010\mathbf{B}$ (Using $B \rightarrow 0B$)
$\rightarrow 00101\mathbf{B}$ (Using $B \rightarrow 1B$)
$\rightarrow 00101$ (Using $B \rightarrow \in$)

2. Rightmost Derivation-

$S \rightarrow A1\mathbf{B}$
$\rightarrow A10\mathbf{B}$ (Using $B \rightarrow 0B$)
$\rightarrow A101\mathbf{B}$ (Using $B \rightarrow 1B$)
$\rightarrow \mathbf{A}101$ (Using $B \rightarrow \in$)
$\rightarrow 0\mathbf{A}101$ (Using $A \rightarrow 0A$)
$\rightarrow 00\mathbf{A}101$ (Using $A \rightarrow 0A$)
$\rightarrow 00101$ (Using $A \rightarrow \in$)

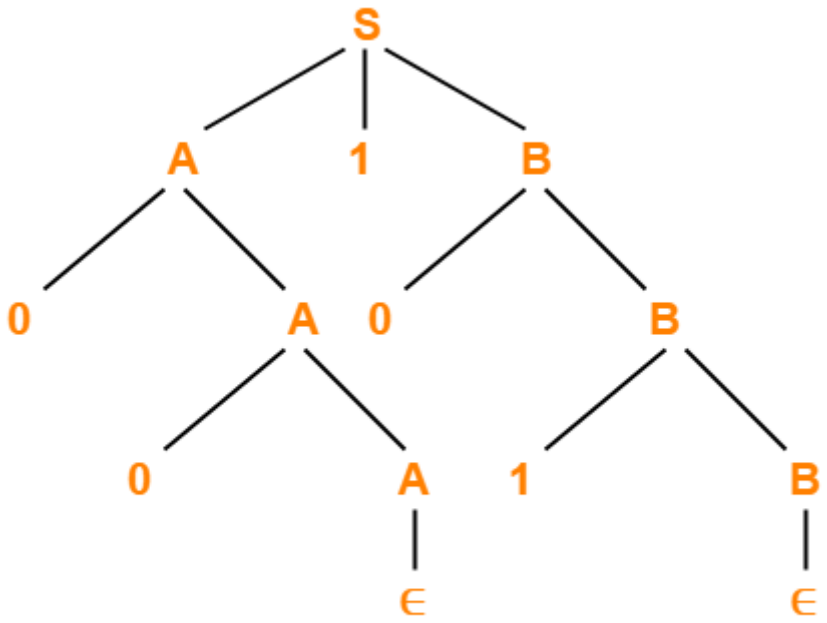
Practice Session Solution

w = 00101

$$S \rightarrow A1B$$

$$A \rightarrow 0A / \in$$

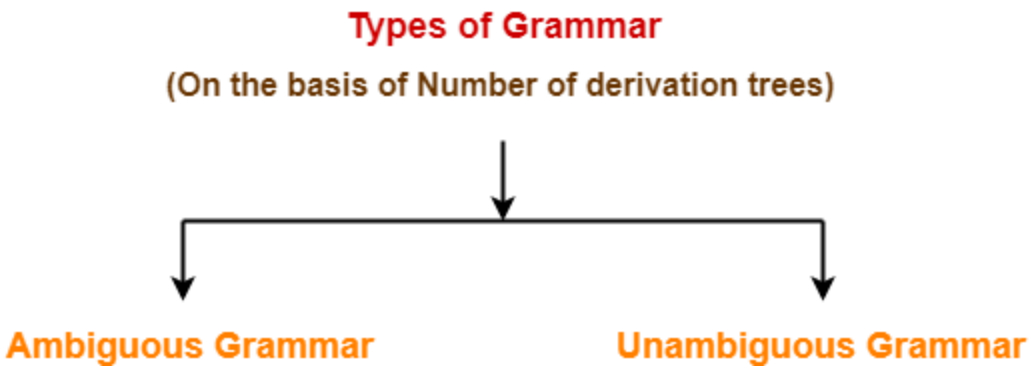
$$B \rightarrow 0B / 1B / \in$$



Parse Tree

Ambiguity in Grammar

- On the basis of number of derivation trees, grammars are classified as-



1.Ambiguous Grammar
2.Unambiguous Grammar

Ambiguity in Grammar

1. Ambiguous Grammar-

A grammar is said to be ambiguous if for any string generated by it, it produces more than one-

- Parse tree
- Or derivation tree
- Or syntax tree
- Or leftmost derivation
- Or rightmost derivation

Ambiguity in Grammar

- Consider the following grammar-

  - $E \rightarrow E + E / E \times E / id$

(Ambiguous Grammar)

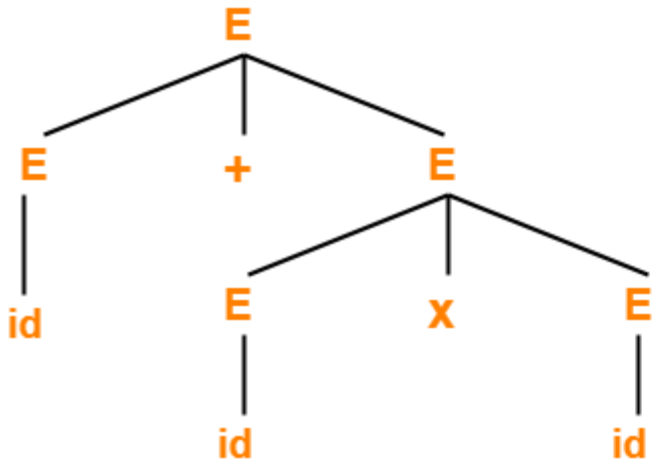
- This grammar is an example of ambiguous grammar.
- Any of the following reasons can be stated to prove the grammar ambiguous-

Ambiguity in Grammar

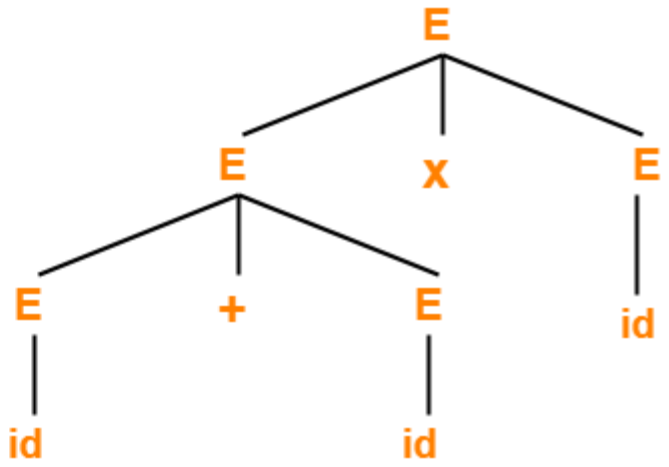
Reason-01:

$E \rightarrow E + E / E \times E / id$

- Let us consider a string w generated by the grammar- w = id + id x id
- Now, let us draw the parse trees for this string w.



Parse Tree-01



Parse Tree-02

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

Ambiguity in Grammar

Reason-02:

- Let us consider a string w generated by the grammar-

$$w = id + id \times id$$

- Now, let us write the leftmost derivations for this string w.

Leftmost Derivation-01

$E \rightarrow \mathbf{E} + E$

$\rightarrow id + \mathbf{E}$

$\rightarrow id + \mathbf{E} \times E$

$\rightarrow id + id \times \mathbf{E}$

$\rightarrow id + id \times id$

Leftmost Derivation-02

$E \rightarrow \mathbf{E} \times E$

$\rightarrow \mathbf{E} + E \times E$

$\rightarrow id + \mathbf{E} \times E$

$\rightarrow id + id \times \mathbf{E}$

$\rightarrow id + id \times id$

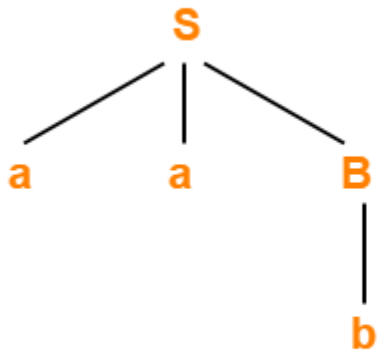
Since two leftmost derivations exist for string w, therefore the grammar is ambiguous.

Ambiguity in Grammar

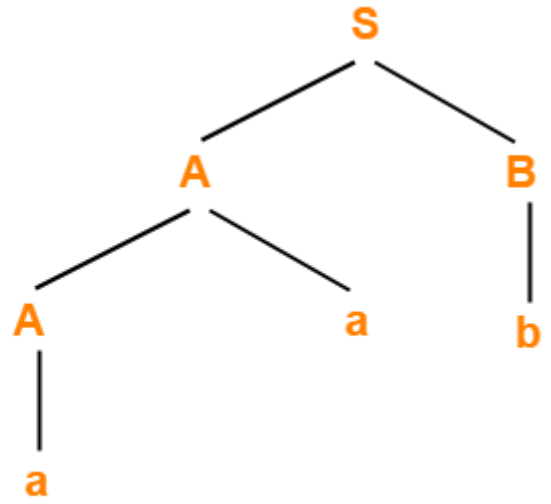
Check whether the given grammar is ambiguous or not-

$$S \rightarrow AB \mid aaB$$
$$A \rightarrow a \mid Aa$$
$$B \rightarrow b$$

Let us consider a string w generated by the given grammar w = aab
Now, let us draw parse trees for this string w.



Parse tree-01

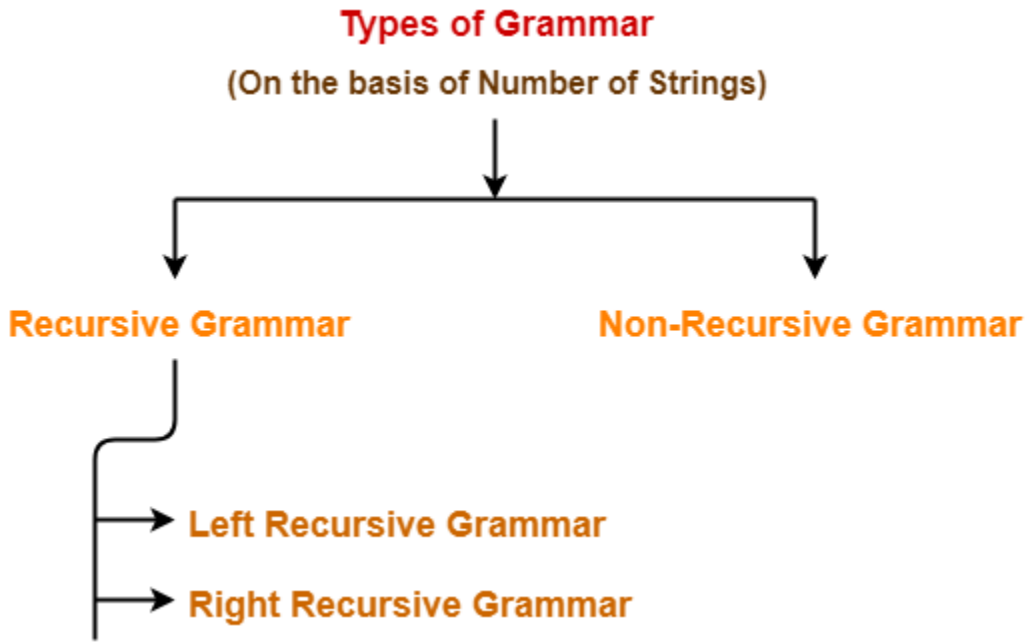


Parse tree-02

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

Recursive Grammar

On the basis of number of strings, grammars are classified as-



1.Recursive Grammar
2.Non-Recursive Grammar

Recursive Grammar

Recursive Grammar- A grammar is said to be recursive if it contains at least one production that has the same variable at both its LHS and RHS.

Or

A grammar is said to be recursive if and only if it generates infinite number of strings.

1.Left Recursive Grammar-A recursive grammar is said to be left recursive if the leftmost variable of RHS is same as variable of LHS.

Example-

$S \rightarrow Sa / b$

(Left Recursive Grammar)

Left recursion is a problem in top-down parsers because **top down parsers use left-most derivation to derive the required string by using the start symbol of the grammar**. Due to this reason top-down parsers might go into infinite loop with left recursive grammar.

Therefore, left recursion has to be eliminated from the grammar.

Recursive Grammar

2. Right Recursion- A production of grammar is said to have **right recursion** if the rightmost variable of its RHS is same as variable of its LHS.

Example-

$S \rightarrow aS / \in$

(**Right Recursive Grammar**)

Right recursion does not create any problem for the Top down parsers.
Therefore, there is no need of eliminating right recursion from the grammar.

Elimination of Left Recursion

Left recursion is eliminated by converting the grammar into a right recursive grammar.

If we have the left-recursive pair of productions-

$$A \rightarrow A\alpha / \beta$$

(Left Recursive Grammar)

where β does not begin with an A.

Then, we can eliminate left recursion by replacing the pair of productions with-

- $A \rightarrow \beta A'$
- $A' \rightarrow \alpha A' / \in$

(Right Recursive Grammar)

This right recursive grammar functions same as left recursive grammar.

Elimination of Left Recursion

Consider the following grammar and eliminate left recursion-

- $E \rightarrow E + T / T$
- $T \rightarrow T \text{ x } F / F$
- $F \rightarrow id$

Solution:

- $E \rightarrow TE'$ $[\mathbf{A \rightarrow \beta A'}]$
- $E' \rightarrow +TE' / \in$ $[\mathbf{A' \rightarrow \alpha A' / \in}]$
- $T \rightarrow FT'$ $[[\mathbf{A \rightarrow \beta A'}]$
- $T' \rightarrow xFT' / \in$ $[\mathbf{A' \rightarrow \alpha A' / \in}]$
- $F \rightarrow id$

Elimination of Left Recursion

Consider the following grammar and eliminate left recursion-

- S → S0S1S / 01

Solution-

The grammar after eliminating left recursion is-

- S → 01S’
- S’→ 0S1SS’ / ∈

Practice Session

Consider the following grammar and eliminate left recursion-

- S → (L) / a
- L → L , S / S

Solution:

The grammar after eliminating left recursion is-

- S → (L) / a
- L → SL’
- L’ → ,SL’ / ∈

Elimination of Left Recursion

The general form for left recursion is

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

can be replaced by

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

Elimination of Left Recursion

Consider the following grammar and eliminate left recursion-

- E → E + E / E x E / a [**A → Aα1/ Aα2/ β**]

- **Solution-**

The grammar after eliminating left recursion is-

- E → aE’ [**A → βA’**]
- E’→ +EE’ / xEE’ / ∈ [**A’ → α1A’ / α2A’ /∈**]

Practice Session

Consider the following grammar and eliminate left recursion-

- S → A
- A → Ad / Ae / aB / ac **[A →Aα1/Aα2/ β1/ β2]**
- B → bBc / f

- Solution:

The grammar after eliminating left recursion is-

- S → A
- A → aBA’ / acA’ **[A → β1A’/ β2A’]**
- A’ → dA’ / eA’ / ∈ **[A → α1A’/α2A’/ ∈]**
- B → bBc / f

End

Theory of Computing
Lec-1

Automata

- **Automata** theory (also known as **Theory Of Computation**) is a theoretical branch of Computer Science and Mathematics, which mainly deals with the logic of computation with respect to simple machines, referred to as automata.

- Automata enables scientists to understand how machines compute functions and solve problems.

- Automata originated from the word “Automaton” which is closely related to “Automation”.

Basic terminologies that are important and frequently used in automata

Symbols, Alphabets

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. ∑ = {A, B, C, D}
3. ∑ = {0, 1, 2}

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 ∑{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:

w = 010

Number of Sting |w| = 3

Language

Language:

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**

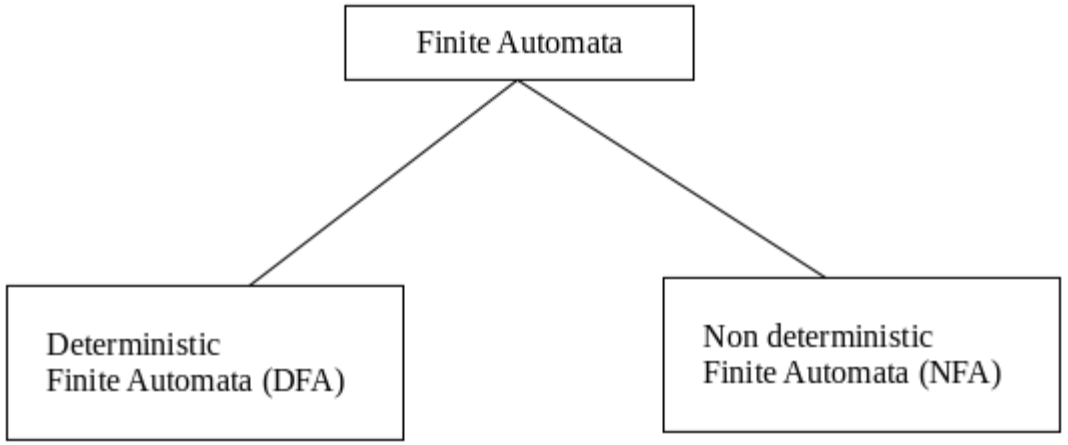
Three basic concepts

- Alphabet- a set of symbols
- Strings-a sequence of symbols from an alphabet
- Language -a set of strings from the same alphabet

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.

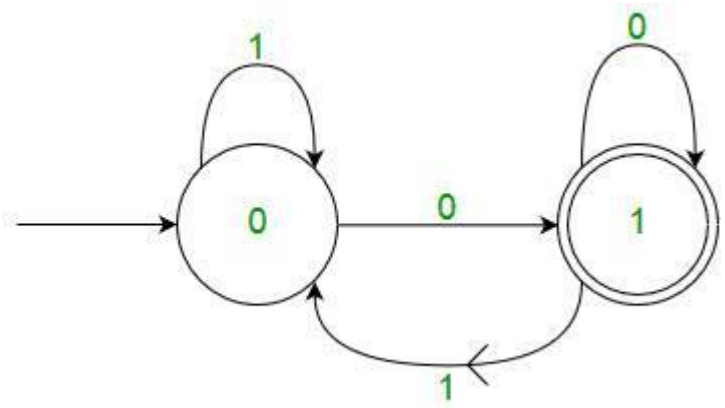
Types of Finite Automata



DFA

- In a DFA, for a particular input character, the machine goes to one state only.
- A transition function is defined on every state for every input symbol. Also in DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character.

For example, below DFA with Σ = {0, 1} accepts all strings ending with 0.



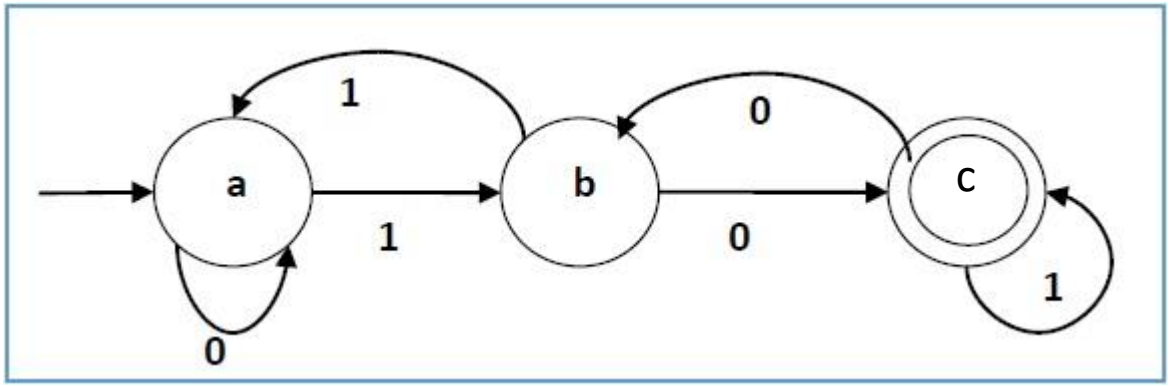
DFA

Formal Definition of DFA

A finite automaton is a collection of 5-tuple (Q, ∑, δ, q0, F), where:

Q: finite set of states
∑: finite set of the input symbol
q0: initial state
F: final state
δ: Transition function

Formal Definition of DFA



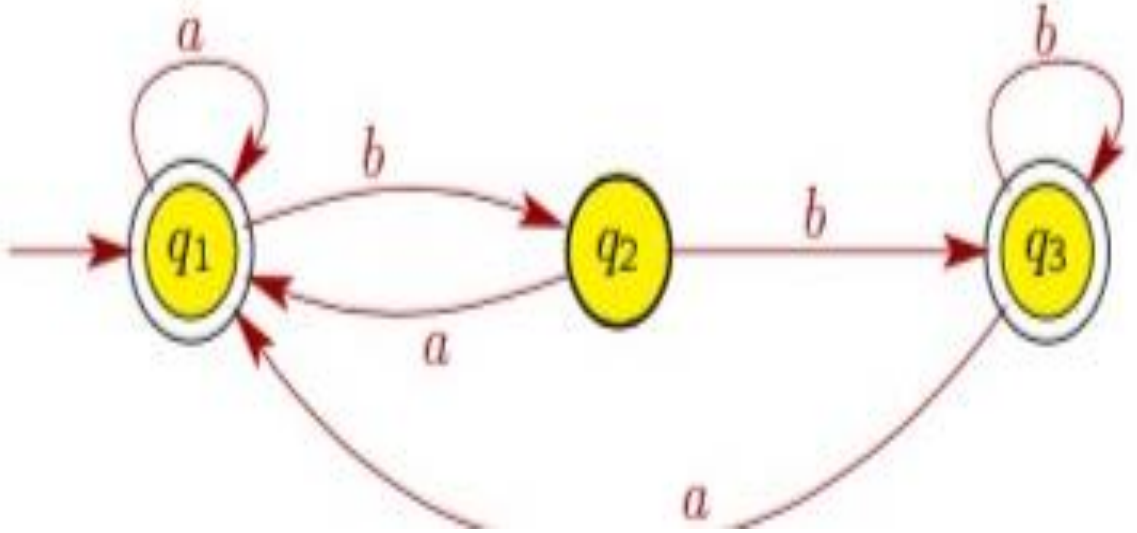
1. Q = {a, b, c},
2. ∑ = {0, 1},
3. q_0 = {a},
4. F = {c} and

5.Transition function δ (Q X Σ --> Q) as shown by the following table

Present State	Next State for Input 0	Next State for Input 1
a	a	b
b	c	a
c	b	c

Exercise

1. For the DFA M below, give its formal definition as a 5-tuple.



Formal Definition of NFA

Nondeterministic Finite Automata(NFA):

NFA refers to Nondeterministic Finite Automaton. A Finite Automata(FA) is said to be non-deterministic if there is more than one possible transition from one state on the same input symbol.

c

A non-deterministic finite automata is also a set of five tuples and represented as,

$$M = \{Q, \Sigma, \delta, qo, F\}$$

Where,

Q: A set of non empty finite states.

Σ: A set of non empty finite input symbols.

δ: It is a transition function that takes a state from Q and an input symbol from and returns a subset of Q.

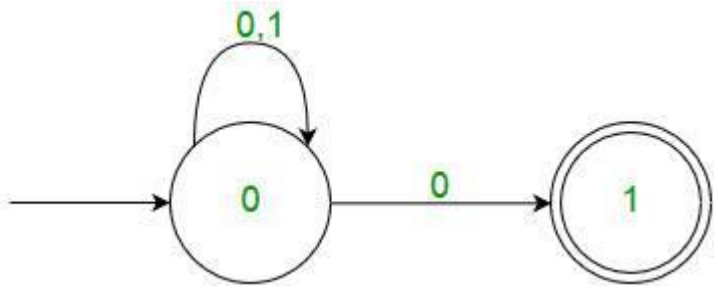
qo: Initial state of NFA and member of Q.

F: A non-empty set of final states and member of Q.

NFA

NFA is similar to DFA except following additional features:

1.Null (or ε) move is allowed i.e., it can move forward without reading symbols.
2.Ability to transmit to any number of states for a particular input.



NFA

One important thing to note is, ***in NFA, if any path for an input string leads to a final state, then the input string* is *accepted***. For example, in the above NFA, there are multiple paths for the input string “00”. Since one of the paths leads to a final state, “00” is accepted by the above NFA.

DFA vs NFA

Difference between DFA and NFA :

DFA	NFA
DFA stands for Deterministic Finite Automata.	NFA stands for Nondeterministic Finite Automata.
For each symbolic representation of the alphabet, there is only one state transition in DFA.	No need to specify how does the NFA react according to some symbol.
DFA cannot use Empty String transition.	NFA can use Empty String transition.
DFA can be understood as one machine.	NFA can be understood as multiple little machines computing at the same time.
In DFA, the next possible state is distinctly set.	In NFA, each pair of state and input symbol can have many possible next states.
DFA is more difficult to construct.	NFA is easier to construct.
DFA rejects the string in case it terminates in a state that is different from the accepting state.	NFA rejects the string in the event of all branches dying or refusing the string.
Time needed for executing an input string is less.	Time needed for executing an input string is more.
All DFA are NFA.	Not all NFA are DFA.
DFA requires more space.	NFA requires less space then DFA.

- What is one difference between NFA and DFA?

Ans: In NFA, each input symbol has zero or more transitions from any state. In DFA, however, each input symbol has precisely one transition from each state.

- Every dfa is nfa. justify this statement. ***

Assignment-1

- Every DFA is NFA. justify this statement. ***

Theory of Computing
Lec-2

Type-1 Problems

In Type-01 problems, we will discuss the construction of DFA for languages consisting of strings ending with a particular substring.

Steps To Construct DFA-

Following steps are followed to construct a DFA for Type-01 problems-

Step-01:

- Determine the minimum number of states required in the DFA.
- Draw those states.

Use the following rule to determine the minimum number of states-

RULE

Calculate the length of substring.

All strings ending with ‘n’ length substring will always require minimum (n+1) states in the DFA.

Step-02:

- Decide the strings for which DFA will be constructed.

Step-03:

- Construct a DFA for the strings decided in Step-02.

Remember the following rule while constructing the DFA-

RULE

While constructing a DFA,

- Always prefer to use the existing path.
- Create a new path only when there exists no path to go with.

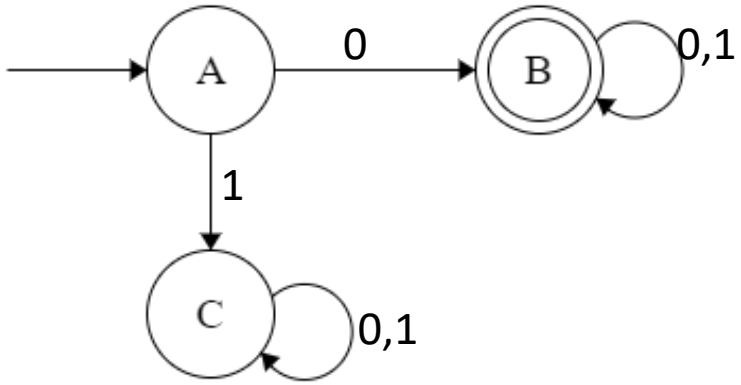
Step-04:

- Send all the left possible combinations to the starting state.
- Do not send the left possible combinations over the dead state.

DFA (Example-1)

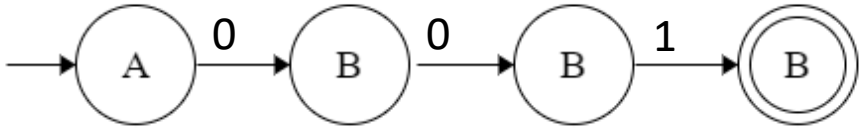
❖ L1=Set of all string that start with 0

Possible cases = {0,00,01,000, 001 ……}

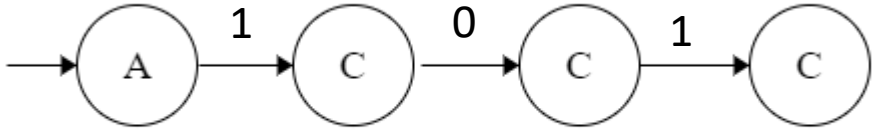


Dead or trap state

E.g. 001



E.g 101



DFA (Example-2)

❖ Draw a DFA for the language accepting strings ending with ’01’ over input alphabets ∑ = {0, 1}

Solution-

Regular expression for the given language = (0 + 1)*01

Step-01:

•All strings of the language ends with substring “01”.
•So, length of substring = 2.

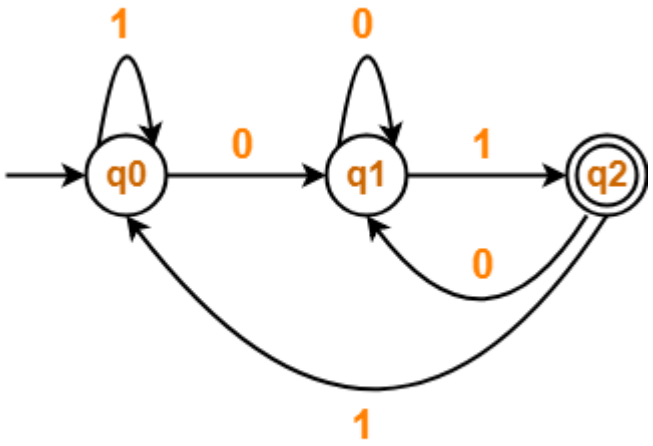
Thus, Minimum number of states required in the DFA = 2 + 1 = 3.
It suggests that minimized DFA will have 3 states.

Step-02:

We will construct DFA for the following strings-
01
001
0101

Step-03:

The required DFA is-



DFA

DFA (Example-3)

❖ Draw a DFA for the language accepting strings ending with ‘abb’ over input alphabets ∑ = {a, b}

Solution-

Regular expression for the given language = (a + b)*abb

Step-01:

All strings of the language ends with substring “abb”.

So, length of substring = 3.

Thus, Minimum number of states required in the DFA = 3 + 1 = 4.

It suggests that minimized DFA will have 4 states.

DFA (Example-3)

Step-02:

We will construct DFA for the following strings-

abb

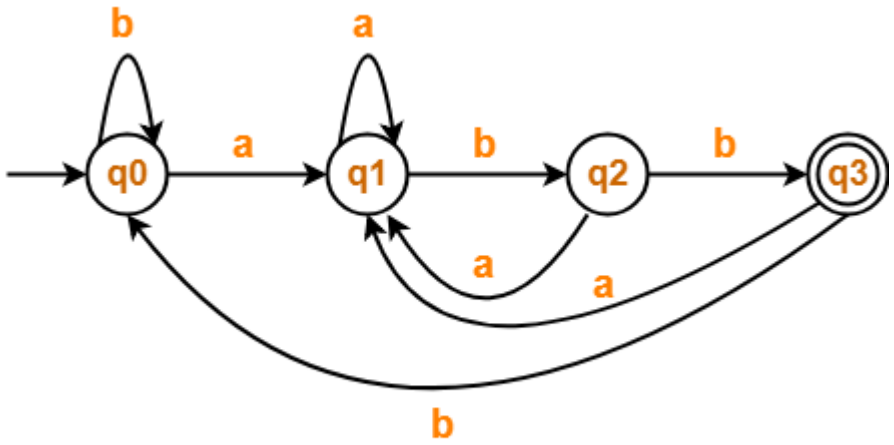
aabb

ababb

abbabb

Step-03:

The required DFA is-



DFA

Practice Session

Draw a DFA for the language accepting strings ending with ‘abba’ over input alphabets ∑ = {a, b}

Practice Session Solution

❖ **Draw a DFA for the language accepting strings ending with 'abba' over input alphabets ∑ = {a, b}**

We will construct DFA for the following strings-

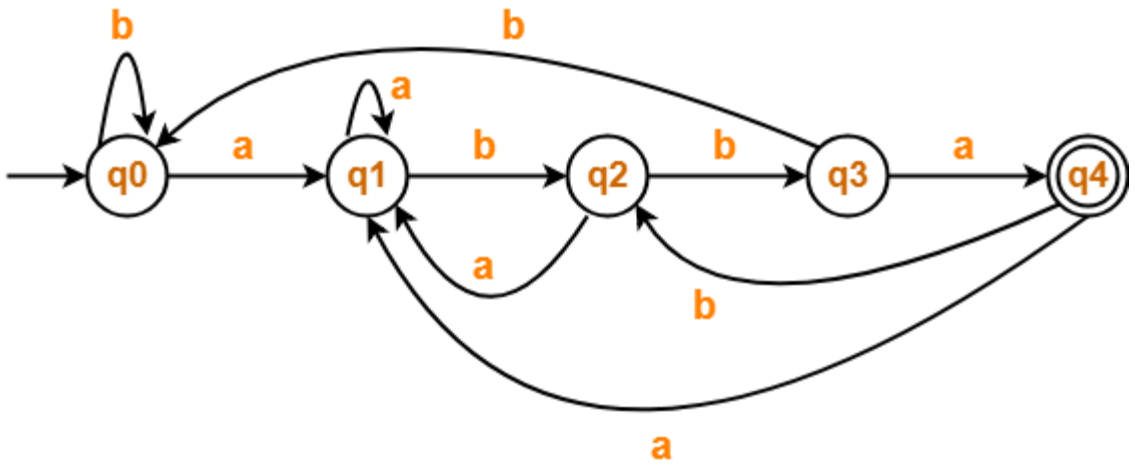
abba

aabba

ababba

abbabba

abbaabba



DFA

Type-2 Problems

❖ Draw a DFA for the language accepting strings starting with ‘a’ over input alphabets ∑ = {a, b}

Solution-

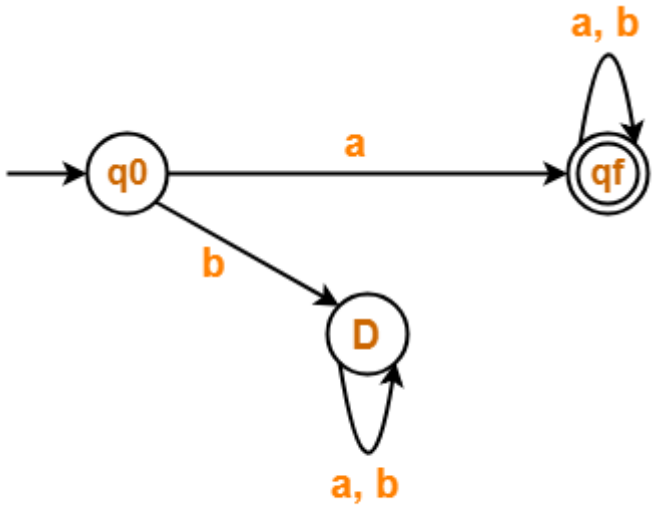
Regular expression for the given language = a(a + b)*

The required DFA is-

We will construct DFA for the following strings-

a

aa



DFA

Type-2 Problems

In Type-02 problems, we will discuss the construction of DFA for languages consisting of strings starting with a particular substring.

❖ Draw a DFA for the language accepting strings starting with ‘ab’ over input alphabets ∑ = {a, b}

Solution-

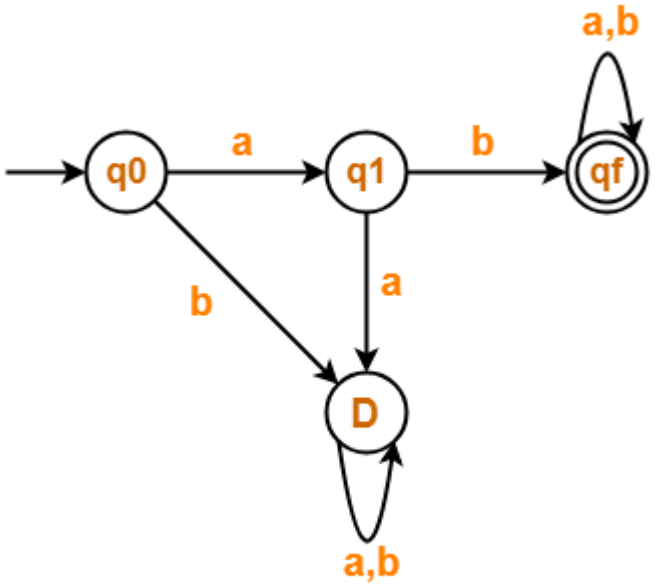
Regular expression for the given language = ab(a + b)*

We will construct DFA for the following strings-

ab

aba

abab



DFA

Practice Session

Draw a DFA for the language accepting strings starting with ‘101’ over input alphabets $\sum = \{0, 1\}$

Practice Session Solution

❖ **Draw a DFA for the language accepting strings starting with '101' over input alphabets ∑ = {0, 1}**

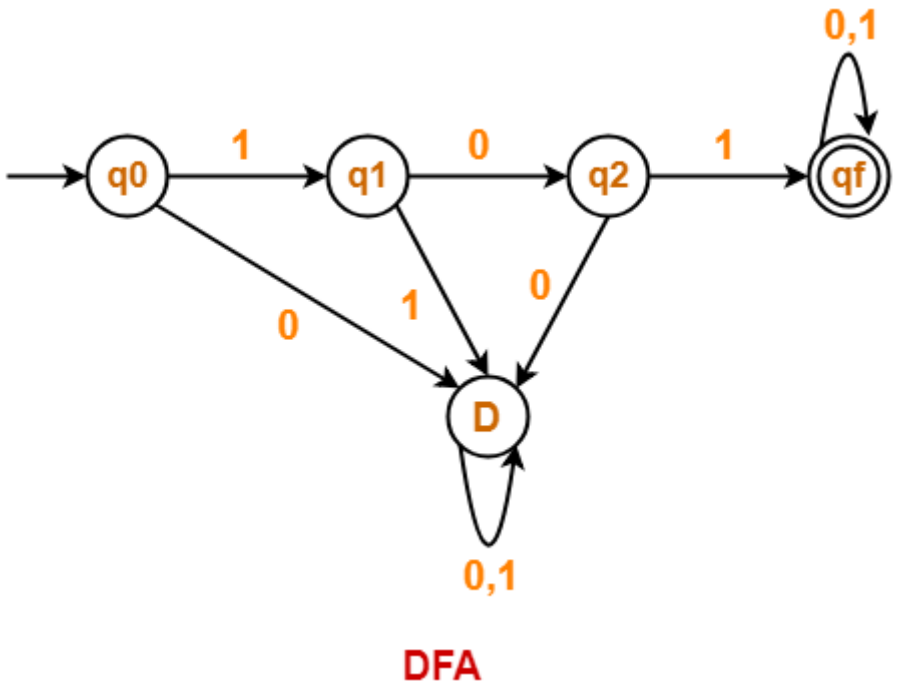
We will construct DFA for the following

101

1011

10110

101101

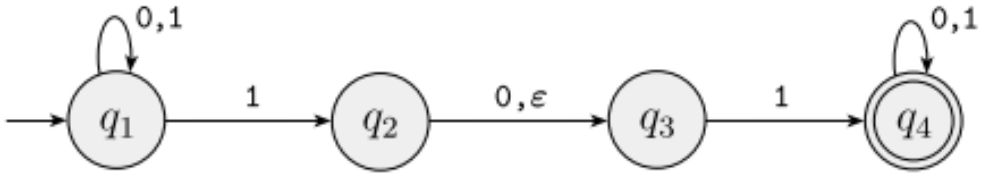


DFA

End

Theory of Computing
Lec-3

NFA Formal Definition



The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0,1\}$,
3. δ is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$,

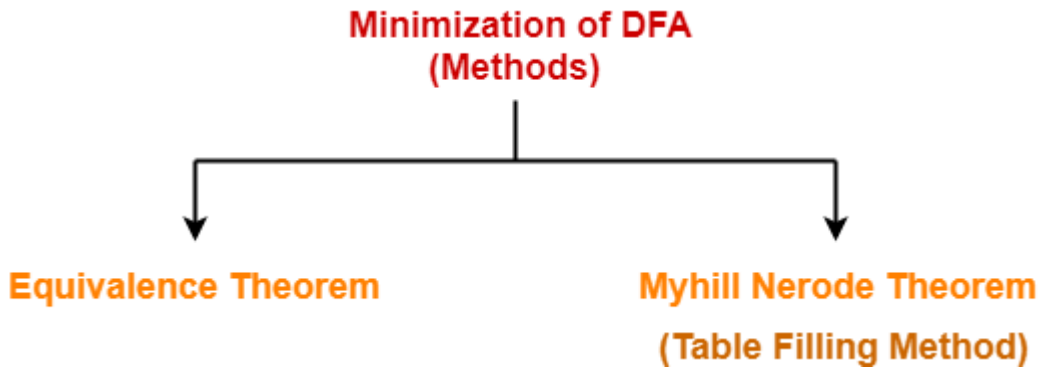
4. q_1 is the start state, and
5. $F = \{q_4\}$.

Minimization of DFA

- The process of reducing a given DFA to its minimal form is called as minimization of DFA.
- It contains the minimum number of states.
- The DFA in its minimal form is called as a Minimal DFA.

How To Minimize DFA?

The two popular methods for minimizing a DFA are-



Minimization of DFA

Minimization of DFA Using Equivalence Theorem-

Step-01:

- Eliminate all the dead states and inaccessible states from the given DFA (if any).

Dead State

All those non-final states which transit to itself for all input symbols in ∑ are called as dead states.

Inaccessible State

All those states which can never be reached from the initial state are called as inaccessible states.

Minimization of DFA

Step-02:

- Draw a state transition table for the given DFA.
- Transition table shows the transition of all states on all input symbols in Σ

Step-03:

- Now, start applying equivalence theorem.
- Take a counter variable k and initialize it with value 0.
- Divide Q (set of states) into two sets such that one set contains all the non-final states and other set contains all the final states.
- This partition is called P_0.

Minimization of DFA

Step-04:

- Increment k by 1.
- Find P_k by partitioning the different sets of P_{k-1} .
- In each set of P_{k-1} , consider all the possible pair of states within each set and if the two states are distinguishable, partition the set into different sets in P_k.

Two states q_1 and q_2 are distinguishable in partition P_k for any input symbol ‘a’,
if $\delta (q_1, a)$ and $\delta (q_2, a)$ are in different sets in partition P_{k-1}.

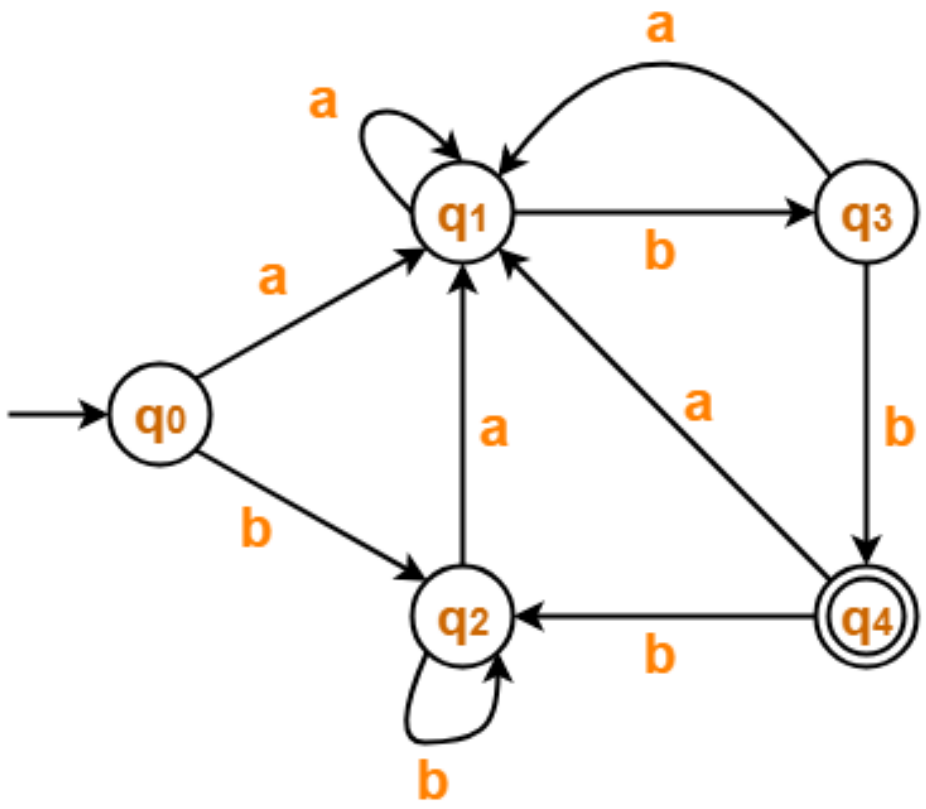
Minimization of DFA

Step-05:

- Repeat step-04 until no change in partition occurs.
- In other words, when you find $P_k = P_{k-1}$, stop.

DFA Minimization Problem-1

Minimize the given DFA-



DFA Minimization Problem-1

Step-01: The given DFA contains no dead states and inaccessible states.

Step-02: Draw a state transition table-

	a	b
→q0	q1	q2
q1	q1	q3
q2	q1	q2
q3	q1	*q4
*q4	q1	q2

Step-03: Now using Equivalence Theorem, we have-

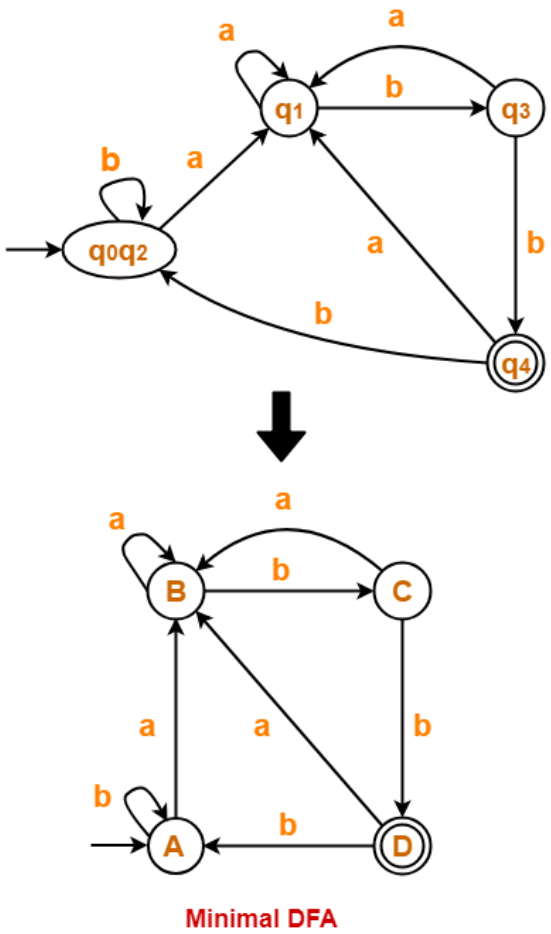
- $P_0 = \{ q_0 , q_1 , q_2 , q_3 \} \{ q_4 \}$
- $P_1 = \{ q_0 , q_1 , q_2 \} \{ q_3 \} \{ q_4 \}$
- $P_2 = \{ q_0 , q_2 \} \{ q_1 \} \{ q_3 \} \{ q_4 \}$
- $P_3 = \{ q_0 , q_2 \} \{ q_1 \} \{ q_3 \} \{ q_4 \}$

Since $P_3 = P_2$, so we stop.

- From P_3, we infer that states q_0 and q_2 are equivalent and can be merged together.

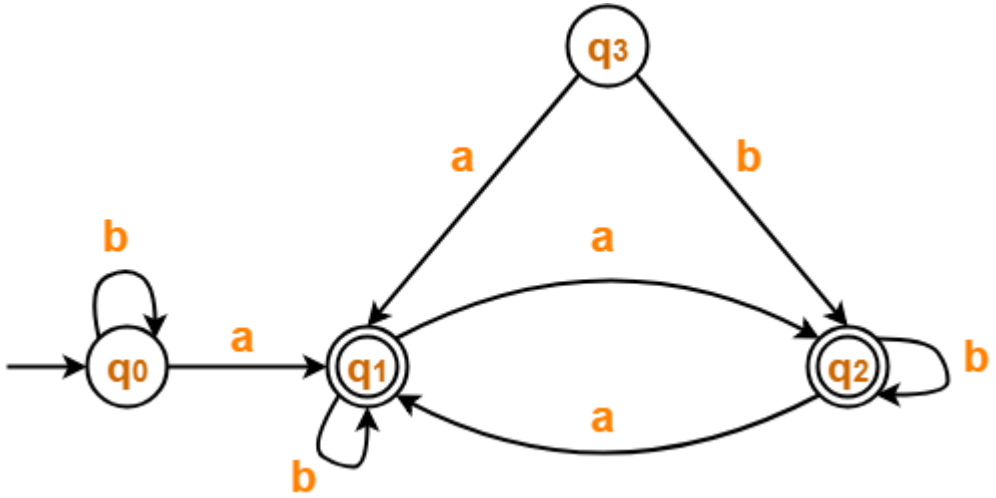
DFA Minimization Solution

So, Our minimal DFA is-



DFA Minimization Problem-2

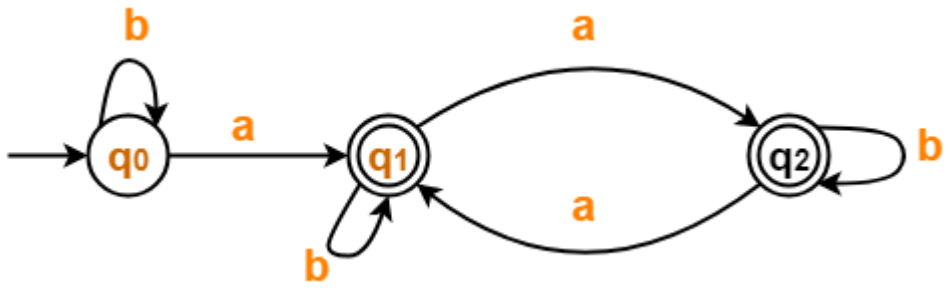
Minimize the given DFA-



DFA Minimization Problem-2

Step-01:

- State q_3 is inaccessible from the initial state.
- So, we eliminate it and its associated edges from the DFA.



DFA Minimization Problem-2

Step-02:

- Draw a state transition table-

	a	b
→q0	*q1	q0
*q1	*q2	*q1
*q2	*q1	*q2

DFA Minimization Solution

Step-03:

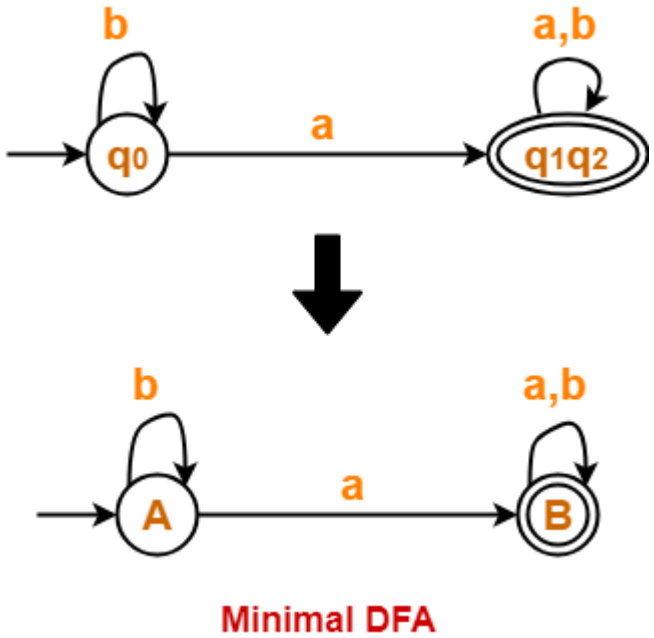
Now using Equivalence Theorem, we have-

- $P_0 = \{ q_0 \} \{ q_1 , q_2 \}$
- $P_1 = \{ q_0 \} \{ q_1 , q_2 \}$

Since $P_1 = P_0$, so we stop.

- From P_1, we infer that states q_1 and q_2 are equivalent and can be merged together.

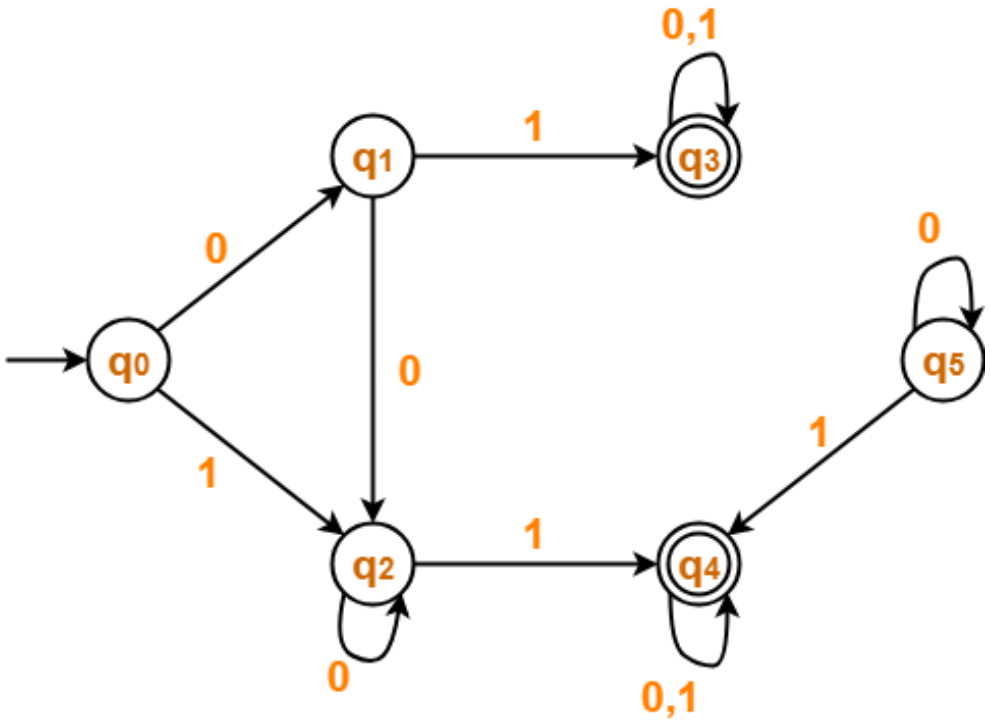
So, Our minimal DFA is-



Minimal DFA

Practice Session

Minimize the given DFA-

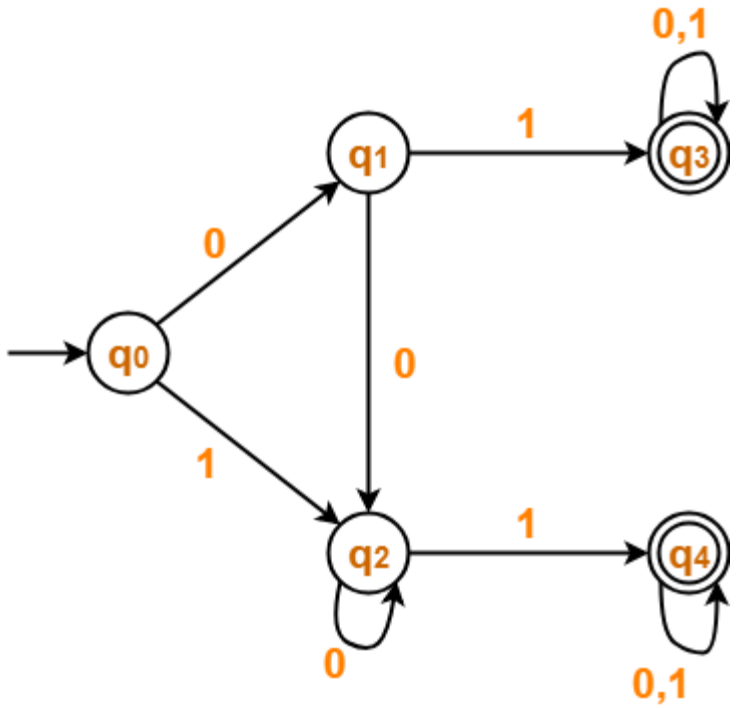


Practice Session Solution

Step-01:

- State q_5 is inaccessible from the initial state.
- So, we eliminate it and its associated edges from the DFA.

The resulting DFA is-



Practise Session Solution

Step-02: Draw a state transition table-

Step-03:Now using Equivalence Theorem, we have-

- $P_0 = \{ q_0 , q_1 , q_2 \} \{ q_3 , q_4 \}$
- $P_1 = \{ q_0 \} \{ q_1 , q_2 \} \{ q_3 , q_4 \}$
- $P_2 = \{ q_0 \} \{ q_1 , q_2 \} \{ q_3 , q_4 \}$

Since $P_2 = P_1$, so we stop.

From P_2, we infer-

- States q_1 and q_2 are equivalent and can be merged together.
- States q_3 and q_4 are equivalent and can be merged together.

	0	**1**
→q0	q1	q2
q1	q2	*q3
q2	q2	*q4
***q3**	*q3	*q3
***q4**	*q4	*q4

Practise Session Solution

So, Our minimal DFA is-