Paper: Formal Languages and Automata Theory
Code: CS403
Contacts: 3L
Credits: 3
Total Lectures: 35

Objective(s)

- Be able to construct finite state machines and the equivalent regular expressions.
- Be able to prove the equivalence of languages described by finite state machines and regular expressions.
- Be able to construct pushdown automata and the equivalent context free grammars.
- Be able to prove the equivalence of languages described by pushdown automata and context free grammars.
- Be able to construct Turing machines and Post machines.
- Be able to prove the equivalence of languages described by Turing machines and Post machines

Outcome(s)

- To acquire the knowledge of the basics of state machines with or without output and its different classifications
- To understand synchronous sequential circuits as the foundation of digital system.
- To apply techniques of designing grammars and recognizers for several programming languages.
- To analyze Turing's Hypothesis as a foreword to algorithms.
- To perceive the power and limitation of a computer, and take decisions on computability.

Prerequisites:

1. Digital Logic
2. Computer organization
3. Computer Fundamentals

Module-1: [9 L]

Fundamentals: Basic definition of sequential circuit, block diagram, mathematical representation, concept of transition table and transition diagram (Related to Automata concept of sequential circuit concept) Design of sequence detector **[2L]**
Introduction to Finite State Model (FSM), Finite State Machine, Finite Automata, Deterministic Finite Automation (DFA) and Non-deterministic Finite Automation (NFA), Transition diagrams, Transition tables and Language recognizers. **[2L]**
NFA with empty transitions, Equivalence between NFA with and without empty transitions. NFA to DFA conversion. **[2L]**
Minimization of FSM: Minimization Algorithm for DFA, Myhill-Nerode Theorem (proof not required) **[2L]**
Limitations of FSM, Application of Finite Automata **[1L]**

Module-2: [7 L]

Finite Automata with output − Moore & Mealy machine. Representation of Moore & Mealy Machine, Processing of the String through Moore & Mealy Machine, Equivalence of Moore & Mealy Machine – Inter-conversion. **[2L]**
Equivalent states and Distinguishable States, Equivalence and k-equivalence, Minimization of Mealy Machine **[1L]**
Minimization of incompletely specified machine − Merger Graph, Merger Table, Compatibility Graph **[2L]**
Lossless and Lossy Machine – Testing Table, Testing Graph **[2L]**

Module-3: [5 L]

Regular Languages, Regular Sets, Regular Expressions, Algebraic Rules for Regular Expressions, Arden's Theorem statement and proof **[1L]**
Constructing Finite Automata (FA) for given regular expressions, Regular string accepted by FA **[2L]**
Constructing Regular Expression for a given Finite Automata **[1L]**
Pumping Lemma of Regular Sets. Closure properties of regular sets (proofs not required). **[1L]**

Module-4: [9 L]

Grammar Formalism - Context Free Grammars, Derivation trees, sentential forms. Right most and leftmost derivation of strings, Parse Tree, Ambiguity in context free grammars. **[1L]**
Minimization of Context Free Grammars. **[1L]**
Chomsky normal form and Greibach normal form. **[1L]**
Pumping Lemma for Context Free Languages. **[1L]**
Enumeration of properties of CFL (proofs omitted). Closure property of CFL, Ogden's lemma & its applications **[1L]**

Regular grammars − right linear and left linear grammars **[1L]**

Push down Automata: Push down automata, definition. Introduction to DCFL, DPDA, NCFL, NPDA **[1L]**
Acceptance of CFL, Acceptance by final state and acceptance by empty state and its equivalence. **[1L]**
Equivalence of CFL and PDA, inter-conversion. (Proofs not required) **[1L]**

Module-5: [5 L]

Turing Machine: Turing Machine, definition, model **[1L]**
Design of TM, Computable functions **[1L]**
Church's hypothesis, counter machine **[1L]**

Types of Turing machines **[1L]**
Universal Turing Machine, Halting problem **[1L]**

TEXT BOOKS:

1. "Introduction to Automata Theory Language and Computation", Hopcroft H.E. and Ullman J. D., Pearson Education.

REFERENCES:

1. "Formal Languages and Automata Theory", C.K.Nagpal, Oxford

2. "Switching & Finite Automata", ZVI Kohavi, 2nd Edition., Tata McGraw Hill

Lesson Plan for B.Tech. Computer Science and Engineering Program (Autonomy)

Paper: Formal Languages and Automata Theory
Code: CS403
Contacts: 3L
Credits: 3
Total Lectures: 35

Module No.	Course Content	Lecture Required	Reference / Text Books
1	Module – 1: [9 L] Fundamentals: Basic definition of sequential circuit, block diagram, mathematical representation, concept of transition table and transition diagram (Related to Automata concept of sequential circuit concept) Design of sequence detector **[2L]** Introduction to Finite State Model (FSM), Finite State Machine, Finite Automata, Deterministic Finite Automation (DFA) and Non-deterministic Finite Automation (NFA), Transition diagrams, Transition tables and Language recognizers. **[2L]** NFA with empty transitions, Equivalence between NFA with and without empty transitions. NFA to DFA conversion. **[2L]** Minimization of FSM: Minimization Algorithm for DFA, Myhill-Nerode Theorem (proof not required) **[2L]** Limitations of FSM, Application of Finite Automata **[1L]**	9 L	**Text Book:** 1. "Introduction to Automata Theory Language and Computation", Hopcroft H.E. and Ullman J. D., Pearson Education. **Reference Book:** 1. "Formal Languages and Automata Theory", C.K.Nagpal, Oxford
2	Module – 2: [7 L] Finite Automata with output − Moore & Mealy machine. Representation of Moore & Mealy Machine, Processing of the String through Moore & Mealy	7 L	**Text Book:** 1. "Introduction to Automata Theory Language and Computation", Hopcroft H.E. and Ullman J. D., Pearson

	Machine, Equivalence of Moore & Mealy Machine – Inter-conversion. **[2L]** Equivalent states and Distinguishable States, Equivalence and k-equivalence, Minimization of Mealy Machine **[1L]** Minimization of incompletely specified machine − Merger Graph, Merger Table, Compatibility Graph **[2L]** Lossless and Lossy Machine – Testing Table, Testing Graph **[2L]**		Education. **Reference Book:** 1. Switching & Finite Automata", ZVI Kohavi, 2nd Edition., Tata McGraw Hill
3	Module – 3: [5 L] Regular Languages, Regular Sets, Regular Expressions, Algebraic Rules for Regular Expressions, Arden's Theorem statement and proof **[1L]** Constructing Finite Automata (FA) for given regular expressions, Regular string accepted by FA **[2L]** Constructing Regular Expression for a given Finite Automata **[1L]** Pumping Lemma of Regular Sets. Closure properties of regular sets (proofs not required). **[1L]**	5 L	**Text Book:** 1. "Introduction to Automata Theory Language and Computation", Hopcroft H.E. and Ullman J. D., Pearson Education. **Reference Book:** 1. "Formal Languages and Automata Theory", C.K.Nagpal, Oxford
4	Module – 4: [9 L] Grammar Formalism - Context Free Grammars, Derivation trees, sentential forms. Right most and leftmost derivation of strings, Parse Tree, Ambiguity in context free grammars. **[1L]** Minimization of Context Free Grammars. **[1L]** Chomsky normal form and Greibach normal form. **[1L]** Pumping Lemma for Context Free Languages. **[1L]** Enumeration of properties of CFL (proofs omitted). Closure property of CFL, Ogden's lemma & its applications **[1L]** Regular grammars − right linear and	9 L	**Text Book:** 1. "Introduction to Automata Theory Language and Computation", Hopcroft H.E. and Ullman J. D., Pearson Education. **Reference Book:** 1. "Formal Languages and Automata Theory", C.K.Nagpal, Oxford

	left linear grammars **[1L]** Push down Automata: Push down automata, definition. Introduction to DCFL, DPDA, NCFL, NPDA **[1L]** Acceptance of CFL, Acceptance by final state and acceptance by empty state and its equivalence. **[1L]** Equivalence of CFL and PDA, inter-conversion. (Proofs not required) **[1L]**		
5	Module – 5: [5 L] Turing Machine: Turing Machine, definition, model **[1L]** Design of TM, Computable functions **[1L]** Church's hypothesis, counter machine **[1L]** Types of Turing machines **[1L]** Universal Turing Machine, Halting problem **[1L]**	5 L	**Text Book:** 1. "Introduction to Automata Theory Language and Computation", Hopcroft H.E. and Ullman J. D., Pearson Education. **Reference Book:** 1. "Formal Languages and Automata Theory", C.K.Nagpal, Oxford

MODULE 1: FINITE AUTOMATA

LECTURE 1: INTREODUCTION TO FINITE AUTOMATA

FINITE AUTOMATA:

A finite automaton can be represented as a 5-tuple structure:

$M = (Q, \Sigma, \delta, q_0, F)$

where,

Q is set of states of the system
Σ is input alphabet of the system
δ is transition function of the system defined as $\delta : Q \times \Sigma \rightarrow Q$
q_0 is the initial or start state of the system
F is the set of final states of the system

Finite Automata are of two types: Non-deterministic Finite Automata (NFA/NDFA) and Deterministic Finite Automata (DFA).

1.1. NON-DETERMINISTIC FINITE AUTOMATA:

A FA is said to be non-deterministic if for a particular input is applied to a current state it is not sure about the next state.

A non-deterministic finite automaton can be represented as a 5-tuple structure:

$M = (Q, \Sigma, \delta, q_0, F)$

where,

Q is set of states of the finite automata
Σ is input alphabet of the finite automata
δ is transition function of the finite automata defined as $\delta : Q \times \Sigma \rightarrow 2^Q$
q_0 is the initial or start state of the finite automata
F is the set of final states of the finite automata

A non-deterministic finite automaton can be denoted by transition diagram or by transition table or by transition function.

Let us take an example of a non-deterministic finite automaton

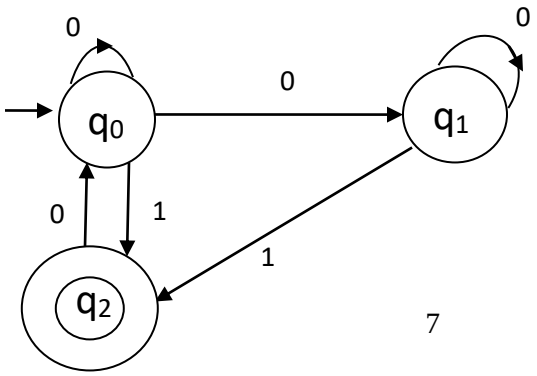


Figure 1.1: Transition Diagram of NFA M1

We can represent the above NFA by a transition table as follows:

PS	NS	
	x=0	**x=1**
→q0	{q0,q1}	{q2}
q1	{q1}	{q2}
*q2	{q0}	-

* Final state

We can represent NFA M1 by 5-tuple structure as follows:

Q = {q0,q1,q2}
Σ = {0,1}
δ(q0,0)={q0,q1}, δ(q0,1)={q2}, δ(q1,0)={q1}, δ(q1,1)={q2}, δ(q2,0)={q0}, δ(q2,1)=Φ
q_0 = q0
F={q2}

1.2. DETERMINISTIC FINITE AUTOMATA:

A FA is said to be non-deterministic if for a particular input is applied to a current state it is very sure about the next state.

A deterministic finite automaton can be represented as a 5-tuple structure:

$M = (Q, \Sigma, \delta, q_0, F)$

where,

Q is set of states of the finite automata
Σ is input alphabet of the finite automata
δ is transition function of the finite automata defined as $\delta : Q \times \Sigma \rightarrow Q$
q_0 is the initial or start state of the finite automata
F is the set of final states of the finite automata

Let us take an example of a deterministic finite automaton

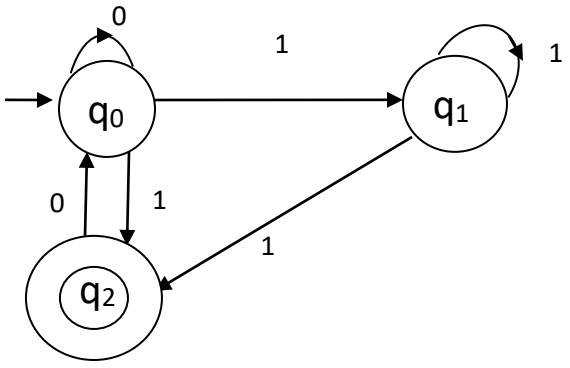


Figure 1.2: Transition Diagram of DFA M2

We can represent the above DFA by a transition table as follows:

PS	NS	
	x=0	**x=1**
→q0	{q0}	{q1}
q1	{q1}	{q2}
*q2	{q0}	-

* Final state

We can represent DFA M2 by 5-tuple structure as follows:

Q = {q0,q1,q2}
Σ = {0,1}
δ(q0,0)={q0}, δ(q0,1)={q1}, δ(q1,0)={q1}, δ(q1,1)={q2}, δ(q2,0)={q0}, δ(q2,1)=Φ
q_0 = q0
F={q2}

1.3. NON-DETERMINISTIC FINITE AUTOMATA WITH empty-Transition

If a FA is modified to permit transition without input symbols, along with zero, one or more transition on input symbols, then we get a NFA with ε-transitions, because the transition made without symbols are called as ε-transitions.

Let us take an example of a non-deterministic finite automaton with empty transitions.

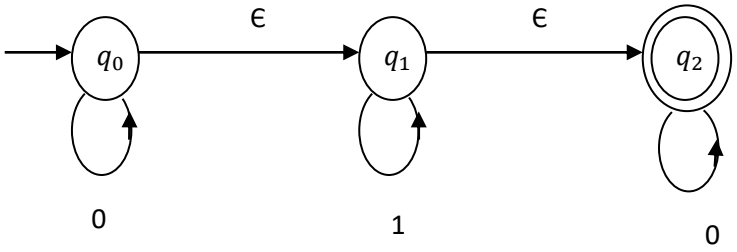


Figure 1.3: Transition Diagram of an NFA with empty transitions M3

Here we define two definitions:
ε-closure of a state, i.e., ε-closure(q), where q is a state, and
ε-closure of a set of states, i.e., ε-closure(q1 ∪ q ∪ 2… ∪ n) , where q1, q2, …, qn are states.

The function **ε-closure(q)** is defined as follows:

ε-closure(q) = set of all those states of the automata (NFA with ε-transitions) which can be reached from q on a path labeled by ε, i.e., without consuming any input symbol.

The function **ε-closure(q1 ∪ q ∪ 2… ∪ n)** is defined as follows:

ε-closure(q1 ∪ q ∪ 2… ∪ n) = ε-closure(q1) ∪ ε-closure(q2) ∪ … ∪ ε-closure(qn)

2. ACCEPTANCE OF STRING BY A FINITE AUTOMATA

2.1. ACCEPTANCE OF STRING BY A DFA

Let us assume the following DFA:

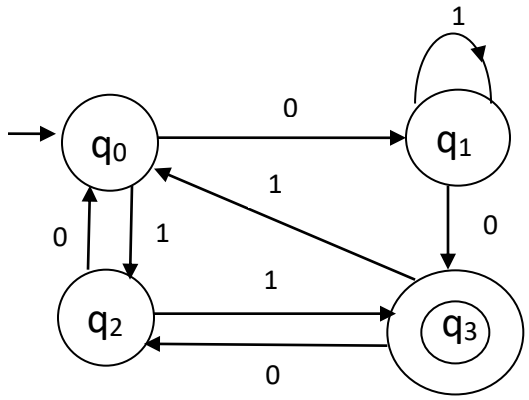
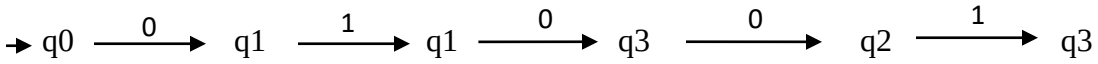


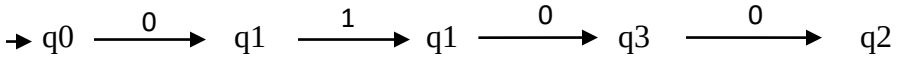
Figure 1.4: Transition Diagram of a DFA M4

For the input string 01001, the transition of states is given by



After processing of entire input string, final state q3 has been reached. So, 01001 is accepted.

For the input string 0100, the transition of states is given by



After processing of entire input string, state q2 has been reached. But q2 is NOT a final state. Hence, 0100 is NOT accepted.

2.2. ACCEPTANCE OF STRING BY AN NFA

Let us assume the following NFA:

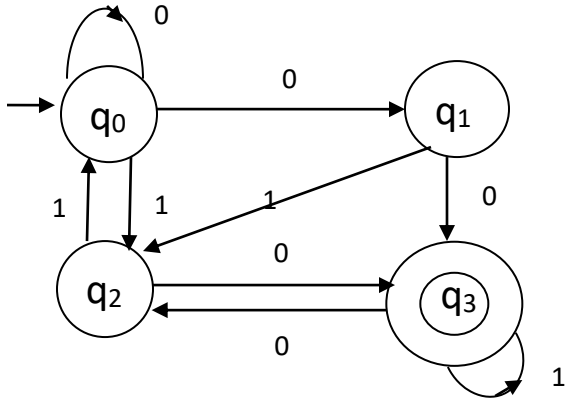
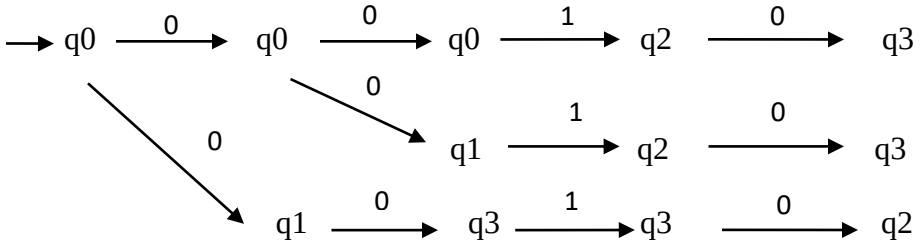


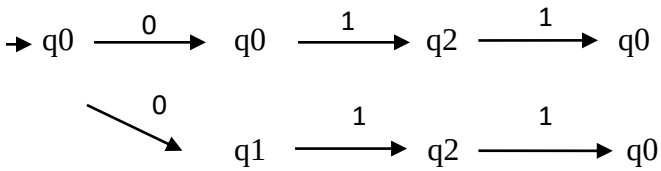
Figure 1.5: Transition Diagram of a NFA M5

For the input string 0010, the transition of states is given by



After processing of entire input string, we find at least one path such that final state q3 has been reached. So, 0010 is accepted.

For the input string 011, the transition of states is given by



After processing of entire input string, final state is NOT reached. Hence, 011 is NOT accepted.

3. EXTENDED TRANSITION FUNCTIONS:

3.1. EXTENDED TRANSITION FUNCTIONS FOR DFA:

The extended transition function is the function that takes a state q and a string w and returns a state P, the state that automation reaches when starting in state q and processing the sequence of inputs w. We define $\hat{\delta}$ by induction on the length of the input string as follows:

Basis: $\hat{\delta}(q,\varepsilon) = q$. That is, if we are in state q and read no input, then we are still in state q.

Induction: Let us suppose w is a string of the form xa, that is a is the last symbol of w and x is the substring of w, consisting of all except the last symbol 'a'.

For example,
w = 1101 is broken into x= 110 and a=1 then,
$\hat{\delta}(q,w)=\delta(\hat{\delta}(q,x),a)$

i.e., to compute $\hat{\delta}(q,w)$, first we compute $\hat{\delta}(q,x)$, the state that the automation is in after processing all but the last symbol of w. Let us suppose this state is P′; that is $\hat{\delta}(q,x) = P'$. Then $\hat{\delta}(q,w)$ is what we get by making a transition from state P′on input a, the last symbol of w.

$\hat{\delta}(q,w) = \delta(P', a) = P$

Let us see this concept through an example.

Let us assume the following DFA:

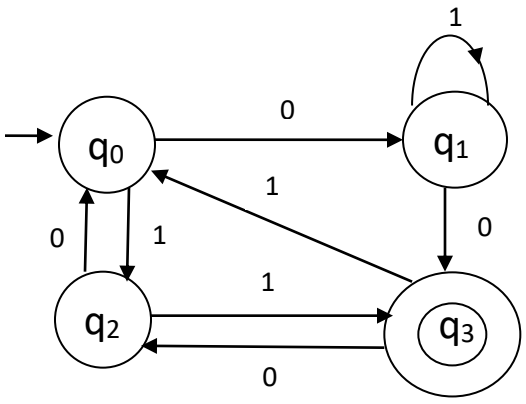


Figure 1.6: Transition Diagram of a DFA M6

For the input string 01001, the transition of states is given by

$\hat{\delta}$ (q0, ε) = q0
$\hat{\delta}$ (q0, 0) = δ ($\hat{\delta}$ (q0, ε), 0) = δ (q0, 0) = q1
$\hat{\delta}$ (q0, 01) = δ ($\hat{\delta}$ (q0, 0), 1) = δ (q1, 1) = q1
$\hat{\delta}$ (q0, 010) = δ ($\hat{\delta}$ (q0, 01), 0) = δ (q1, 0) = q3
$\hat{\delta}$ (q0, 0100) = δ ($\hat{\delta}$ (q0, 010), 0) = δ (q3,0) = q2
$\hat{\delta}$ (q0, 01001) = δ ($\hat{\delta}$ (q0, 0100), 1) = δ (q2, 1) = q3

Final state q3 has been reached. So, 01001 is accepted.

3.2. EXTENDED TRANSITION FUNCTIONS FOR NFA:

The extended transition function is the function that takes a state q and a string w and returns a set of states that automation reaches when starting in state q and processing the sequence of inputs w. We define $\hat{\delta}$ by induction on the length of the input string as follows:

Basis: $\hat{\delta}$ (q,ε) = {q}. That is, if we are in state q and read no input, then we are still in state q.

Induction: Let us suppose w is a string of the form xa, that is a is the last symbol of w and x is the substring of w, consisting of all except the last symbol 'a'.

Let us suppose that

$\hat{\delta}$ (q,x) = {p1, p2, …, pk},
Let,
$\bigcup_{i=1}^{k} \delta(pi, a)$ = {r1, r2, r3, …, rm}

Then,
$\hat{\delta}$ (q,x) = {r1, r2, r3, …, rm}.

Less formally, we compute $\hat{\delta}$ (q, w) by first computing $\hat{\delta}$ (q, x), and by then following any transition from any of these states that is labeled a. Let us see an example for better understanding of the concept.

Let us see this concept through an example.

Let us assume the following NFA:

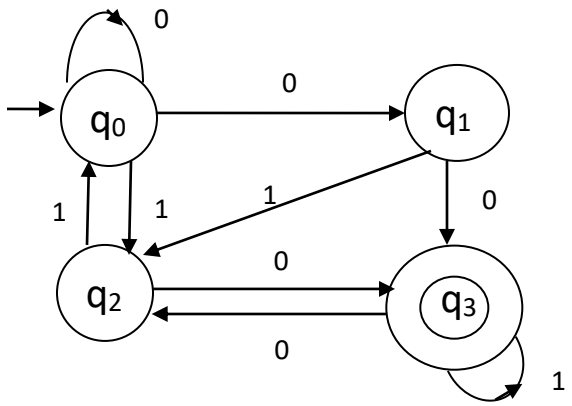


Figure 1.7: Transition Diagram of a NFA M7

For the input string 0010, the transition of states is given by

$\hat{\delta}$ (q0, ε) = {q0}
$\hat{\delta}$ (q0, 0) = {q0,q1}
$\hat{\delta}$ (q0, 00) = δ (q0, 0) ∪ δ (q1, 0) = {q0,q1} ∪ {q3} = {q0,q1,q3}
$\hat{\delta}$ (q0, 001) = δ (q0, 1) ∪ δ (q1, 1) ∪ δ (q3, 1) = {q2} ∪ {q2} ∪ {q3} = {q2,q3}
$\hat{\delta}$ (q0, 0010) = δ (q2,0) ∪ δ (q3,0) = {q3,q2}

Final state q3 may be reached. So, 0010 is accepted.

LECTURE 2: EQUIVALENCE BETWEEN NFA AND DFA

4. CONVERSION:

4.1. CONVERSION: NFA WITH ε-TRANSITIONS TO NFA WITHOUT ε-TRANSITIONS (REMOVING ε-TRANSITIONS)

Let us consider the following NFA M8 with ε-transitions. We have to find an equivalent NFA without ε-transitions.

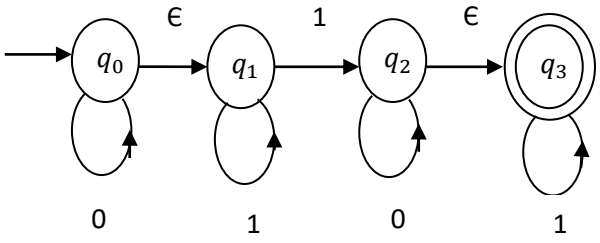


Figure 1.8a: Transition Diagram of a NFA M8

First, we calculate ε-closure of each state. And those will be states of new system.

ε-closure(q_0) = {q_0,q_1}= A
ε-closure(q_1) = {q_1} = B
ε-closure(q_2) = {q_2,q_3} = C
ε-closure(q_3) = {q_3} = D

Initial state of NFA without **ε**-transition will be **ε**-closure of initial state of NFA with **ε**-transition. So, A is start state.

The final states of NFA without **ε**-transition are all those new states which contains final state of NFA with **ε**-transition as member
So, C and D are final states.

So if NFA without **ε**-transition is
M′= (Q′, Σ, δ′,q0′, F′), where

Q′ = {A,B,C}
q0′= A
F′= {C, D}

Now, we have to decide δ′to find out the transitions as follows:

δ′(A,0)
= **ε-closure** (δ(A,0))
= **ε-closure** (δ({q0,q1},0))
= **ε-closure** (δ(q0,0) ∪ δ(q1,0))
= **ε-closure** ({q0} ∪ Φ)
= **ε-closure** ({q0} ∪ Φ)
= **ε-closure** ({q0})
= **ε-closure** (q0)
= {q0,q1}
= A

δ′(A,1)
= **ε-closure** (δ(A,1))
= **ε-closure** (δ({q0,q1},1))
= **ε-closure** (δ(q0,1) ∪ δ(q1,1))
= **ε-closure** (Φ ∪{q1,q2})
= **ε-closure** ({q1,q2})
= **ε-closure** (q1) ∪ **ε-closure** (q2)
= {q1} ∪{q2,q3}
= B∪C
= {B,C}

δ′(B,0)
= **ε-closure** (δ(B,0))
= **ε-closure** (δ({q1},0))
= **ε-closure** (δ(q1,0))
= **ε-closure** (Φ)
= Φ

δ′(B,1)
= **ε-closure** (δ(B,1))
= **ε-closure** (δ({q1},1))
= **ε-closure** (δ(q1,1))
= **ε-closure** ({q1,q2})
= **ε-closure** (q1) ∪ **ε-closure** (q2)
= {q1} ∪{q2,q3}
= B∪C
= {B,C}

δ′(C,0)
= **ε-closure** (δ(C,0))
= **ε-closure** (δ({q2,q3},0))
= **ε-closure** (δ(q2,0) ∪ δ(q3,0))
= **ε-closure** ({q2} ∪ Φ)
= **ε-closure** ({q2})
= **ε-closure** (q2)
= {q2,q3}
= C

δ′(C,1)
= **ε-closure** (δ(C,1))
= **ε-closure** (δ({q2,q3},1))
= **ε-closure** (δ(q2,1) ∪ δ(q3,1))
= **ε-closure** (Φ∪{q3})
= **ε-closure** ({q3})
= **ε-closure** (q3)
= {q3}
= D

δ′(D,0)
= **ε-closure** (δ(D,0))
= **ε-closure** (δ({q3},0))
= **ε-closure** (δ(q3,0))
= **ε-closure** (Φ)
= Φ

δ′(D,1)
= **ε-closure** (δ(D,1))
= **ε-closure** (δ({q3},1))
= **ε-closure** (δ(q3,1))
= **ε-closure** ({q3})
= **ε-closure** (q3)
= {q3}
= D

Now, we can draw the transition diagram of equivalent NFA without ε-transitions as follows:

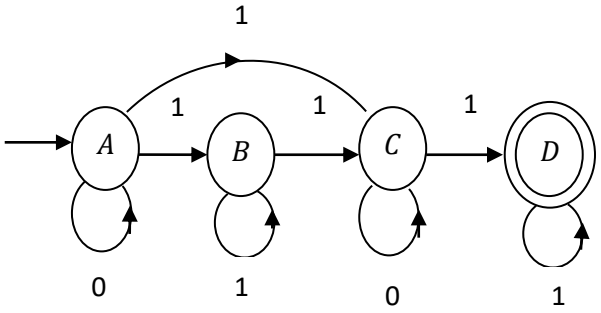


Figure 1.8b: Transition Diagram of an equivalent NFA M8′ of NFA M8

4.2. CONVERSION: NFA WITHOUT ε-TRANSITIONS TO DFA

First we remove Multiple Transitions (if any):

Let M be an NFA denoted by (Q, Σ, δ, q0, F) which accepts L.
To obtain an equivalent DFA M′ = (Q′, Σ, δ′, q0, F′) which accepts the same language as given NFA M = (Q, Σ, δ, q0, F) does, we may proceed as follows:

Step 1: Initially Q′ = Φ.
Step 2: We put [q0] into Q′. [q0] is the initial state of DFA M′.
Step 3: We add every new state q to Q′, where $\delta'(q, a) = \cup_{P \in Q} \delta(P, a)$
Step 4: We repeat step 3 till new states are there to add in Q′, if there is no further new state found to add in Q′, the process terminates. All states in Q′ that contain final states of M are accepting state of M′.

Note: The states which are not reachable from the initial state should not be included in Q′. Thus the set of states (Q′) is not necessarily equal to 2^Q.

Then we remove Undefined Transitions (if any):

If (A,a) is undefined then we incorporate a new state I (known as Idle state or Dead state or Trap state) such that
$\delta(A,a) = I$
and
$\delta(I,x) = I$ for all $x \in \Sigma$

Example:

Let us consider the following NFA M9 without ε-transitions. We have to find an equivalent DFA.

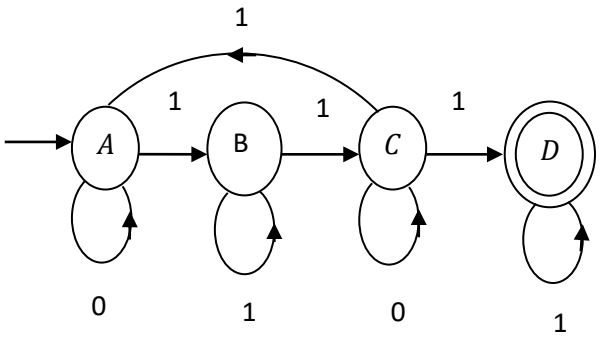


Figure 1.9a: Transition Diagram of NFA M9

Solution:

First we draw the transition table of given NFA M9.

PS	**NS, z**	
	x=0	**x=1**
→ A	{A}	{B,C}
B	-	{B,C}
C	{C}	{D}
*D	-	{D}

First we remove Multiple Transitions:

Step 1: We denote start state A by [A]. We seek all the transition from starting state [A] for every symbol in Σ i.e. (0, 1). If we get a set of states for same input then we consider that set as new single state as:

$\delta([A],0) = \delta([A],0) = \{A\}$
$\delta([A],1) = \delta([A],1) = \{B,C\}$

We denote {A} by [A] and {B,C} by [B,C] in the DFA.

Step 2: In step 1 we are getting a new state [B, C]. Now we repeat step 12 for this new state only, i.e., we check all transitions of 0 and 1 (that is Σ) from [B, C] as:

$\delta([B,C],0) = \delta(B,0) \cup \delta(C,0) = \Phi \cup \{C\} = \{C\}$
$\delta([B,C],1) = \delta(B,1) \cup \delta(C,1) = \{B,C\} \cup \{D\} = \{B,C,D\}$

We denote {C} by [C] and {B,C,D} by [B,C,D] in the DFA.

Step 3: We repeat step 2 till we are getting any new state. All those states which consists at least one accepting state of given NFA as member state will be considered as final states.

[C] is new state.

$\delta([C],0) = \delta(C,0) = \{C\}$
$\delta([C],1) = \delta(C,1) = \{D\}$

We denote {C} by [C] and {D} by [D] in the DFA.

[B,C,D] is new state.

$\delta([B,C,D],0) = \delta(B,0) \cup \delta(C,0) \cup \delta(D,0) = \Phi \cup \{C\} \cup \Phi = \{C\}$
$\delta([B,C,D],1) = \delta(B,1) \cup \delta(C,1) \cup \delta(D,1) = \{B,C\} \cup \{D\} \cup \{D\} = \{B,C,D\}$

We denote {C} by [C] and {B,C,D} by [B,C,D] in the DFA.

[D] is new state.

$\delta([D],0) = \delta(D,0) = \Phi$
$\delta([D],1) = \delta(D,1) = \{D\}$

We denote {C} by [C] and {D} by [D] in the DFA.

Here, [B,C,D] and [d] are final states.

Let us draw the transition table:

PS	**NS, z**	
	x=0	**x=1**
→[A]	[A]	[B,C]
[B,C]	[C]	[B,C,D]
[C]	[C]	[D]
* [B,C,D]	[C]	[B,C,D]
* [D]	-	[D]

Let us replace,

[A] by Q0,
[B,C] by Q1
[C] by Q2
[B,C,D] by Q3
[D] by Q4

Hence the transition table is as follows:

PS	NS, z	
	x=0	x=1
→ Q0	Q0	Q1
Q1	Q2	Q3
Q2	Q2	Q4
*Q3	Q2	Q3
*Q4	-	Q4

Now we remove Undefined Transition:

Here, δ(Q4,0) is undefined.

Hence we incorporate a new state I such that
δ(Q4,0) = I
and
δ(I, 0) = I and δ(I, 1) = I

Hence the transition table of equivalent DFA is as follows:

PS	NS, z	
	x=0	x=1
→ Q0	Q0	Q1
Q1	Q2	Q3
Q2	Q2	Q4
*Q3	Q2	Q3
*Q4	I	Q4
I	I	I

And, the transition diagram of equivalent DFA is as follows:

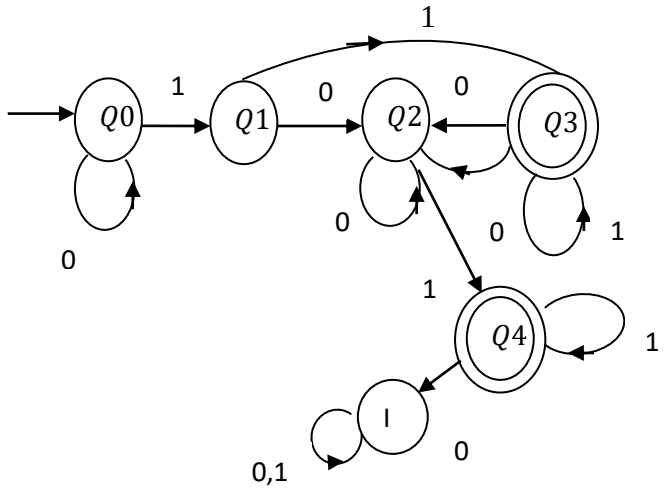


Figure 1.9b: Transition Diagram of an equivalent DFA M9′ of given NFA M9

LECTURE 3: MINIMIZATION TO FINITE AUTOMATA

Definitions of Dead State and Inaccessible State:

Dead State: All those non-final states which transit to itself for all input symbols in Σ, are called Dead state.

Inaccessible State or Unreachable State: All those states which can never be reached from initial states are called inaccessible states or unreachable states.

5. MINIMIZATION OF DFA

5.1. MINIMIZATION OF DFA USING SUBSET CONSTRUCTION METHOD

For any DFA with more number of states we can construct its equivalent DFA with minimum number of states.

Equivalent states: Two states q1 and q2 are said to be equivalent if both $\delta(q1,x)$ and $\delta(q2,x)$ are final states, or both of them are non-final states for all $x \in \Sigma^*$.

As it is difficult to construct $\delta(q1,x)$ and $\delta(q2,x)$ for all $x \in \Sigma^*$ (there are an infinite number of strings in Σ^*), we give one alternate definition.

Two states q1 and q2 are **k-equivalent** ($k \geq 0$) if both $\delta(q1,x)$ and $\delta(q2,x)$ are final states or both non-final states for all strings x of length k or less. In particular, any two final states are 0-equivalent and any two non-final states are also 0-equivalent.

We mention some of the properties of these relations.

Property 1: The relations we have defined, i.e. equivalence and k-equivalence, are equivalence relations, i.e., they are reflexive, symmetric and transitive.

Property 2: These induce partitions of Q. These partitions can be denoted by P and P_K, respectively. The elements of P_K are k-equivalence classes.

Property 3: If q1 and q2 are k-equivalent for all $k \geq 0$, then they are equivalent.

Property 4: If q1 and q2 are (k+1)-equivalent for all $k \geq 0$, then they are k-equivalent.

Property 5: $\Pi_n = \Pi_{n+1}$ for some n. (Π_n denotes the set of equivalence classes under n-equivalence.

The following result is the key to the construction of minimum state automation.

Result:
Two states q1 and q2 are (k+1)-equivalent if
(i) they are k-equivalent
(ii) $\delta(q1,x)$ and $\delta(q2,x)$ are also k-equivalent for every a ε Σ

Construction of Minimum Automation:

0. **Removal of all unreachable states**: First we remove all unreachable states.
1. **Construction of P_0**: Initially we construct 0-equivalence class as P0 = $\{Q_1^0, Q_2^0\}$ where Q_1^0 is set of final states and $Q_2^0 = Q - Q_1^0$ is the set of non-final states.
2. **Construction of P_{K+1} from P_K**:
   (a) Let Q_1^K be any subset in P_K. If q1 and q2 are in Q_1^K they are (K+1) equivalent provided $\delta(q1,a)$ and $\delta(q2,a)$ are K-equivalent.
   (b) We find out whether $\delta(q1,a)$ and $\delta(q2,a)$ are in same equivalence class in P_K for every a ϵ Σ. If so, q1 and q2 are (k+1) equivalent. This way Q_1^K is further divided into (K+1) equivalence classes. This is repeated for every Q_K in P_K to get all the elements of P_{K+1}.
   Putting in another way, two states are placed in the **same block** of P_{K+1} if and only if they are in **the same block** of P_K, and for each possible x ε Σ their x-successors are also contained in **a same** block of P_K. This step is carried out by **splitting blocks** of P_i. In general, the P_{K+1} Partition is obtained from P_K by placing in the same block of P_{K+1} those states which are in the **SAME BLOCK** of P_K and whose x-successors for every possible x ε Σ are also in a **COMMON BLOCK** of P_K.
3. **Construction of P_n for n = 1,2,3, …** : We construct P_n for n = 1,2,3,….. until $P_n = P_{n+1}$.
4. **Construction of minimized DFA**: For the required minimum state automation, states are equivalent classes obtained in step 3, i.e., the elements of P_n.

Example:

Find minimum state automation for the following DFA:

PS	NS	
	x=0	x=1
→ q0	q1	q2
q1	q2	q4
q2	q3	q2
q3	q2	q4
*q4	q1	q4

(a) Initially we identify 0-equivalence as $P_0 = \{Q_1^0, Q_2^0\}$ where Q_1^0 is set of final states and $Q_2^0 = Q - Q_1^0$ is set of non-final states.

$Q_2^0 = \{q4\}$
$Q_2^0 = \{q0, q1, q2, q3\}$

(b) We construct P_1 from P_0 identifying the equivalent states in $\{Q_1^0, Q_2^0\}$

Q_1^0 cannot be divided as it has only one state.

Q_2^0 has four states, we need to identify whether they are 1-equivalent.

We compare q0 and q1 on input 0 and 1

$\delta(q0,0) = q1$
$\delta(q1,0) = q2$ both resultant states belong to Q_2^0
$\delta(q0,1) = q2$
$\delta(q1,1) = q4$ both resultant states belong to different sets in P_0
=> q0 is not 1-equivalent to q1

We compare q0,q2 on input 0 and 1
$\delta(q0,0) = q1$
$\delta(q2,0) = q3$ both resultant states belong to Q_2^0
$\delta(q0,1) = q2$
$\delta(q2,1) = q1$ both resultant states belong to different sets in Q_2^0
=> q0 is not 1-equivalent to q2

We compare q0,q3 on input 0 and 1
$\delta(q0,0) = q1$
$\delta(q3,0) = q2$ both resultant states belong to Q_2^0
$\delta(q0,1) = q2$
$\delta(q3,1) = q4$ both resultant states belong to different sets P_0
=> q0 is not 1-equivalent to q3

Hence,

$P_1 = \{ Q_1^1, Q_2^1, Q_3^1 \}$

where,
$Q_1^1 = \{q4\}$
$Q_2^1 = \{q0,q2\}$
$Q_3^1 = \{q1,q3\}$

(c) We construct P_2 from P_1 identifying the equivalent states in $\{Q_1^1, Q_2^1, Q_3^1\}$

Q_1^1 cannot be divided as it has only one state.

Q_2^1 and Q_3^1 has two states each, we need to identify whether they are equivalent.

We compare q0, q2 on input 0 and 1
$\delta(q0,0) = q1$
$\delta(q2,0) = q3$ both resultant states belong to Q_3^1
$\delta(q0,1) = q2$
$\delta(q2,1) = q1$ both resultant states belong to different sets in Q_2^1
=> q0 is 1-equivalent to q2

We compare q1, q3 on input 0 and 1
$\delta(q1,0) = q2$
$\delta(q3,0) = q2$ both resultant states belong to Q_2^1
$\delta(q1,1) = q4$
$\delta(q3,1) = q4$ both resultant states belong to different sets in Q_1^1
=> q0 is not 1-equivalent to q3

Hence,

$\mathbf{P_2} = \{ Q_1^2, Q_2^2, Q_3^2\}$

where,
$Q_1^2 = \{q4\}$
$Q_2^2 = \{q0,q2\}$
$Q_3^2 = \{q1,q3\}$

(d) Here, we are presenting all Partitions as follows:

P_0 = {{q4},{q0,q1,q2,q3}}
P_1 = {{q4},{q0,q2},{q1,q3}}
P_2 = {{q4},{q0,q2},{q1,q3}}

We see that P2 is equal to P1. The states q0 and q2 are considered as a single state q02, and q1 and q3 as a single state q13. Minimized DFA is:

PS	**NS**	
	x=0	**x=1**
→ q02	q13	q02
q13	q02	q4
*q4	q13	q4

5.2. MINIMIZATION OF DFA USING MYHILL NERODE THEOREM:

Let us think of an equivalence relation as being true or false for a specific pair of strings x and y. Thus xRy is true for some set of pairs x and y. We will use a relation R such that xRy ⇔ yRx, x has a relation to y if and only if y has the same relation to x. This is known as symmetric.

xRy and yRz => xRz. This is known as transitive.
xRx is true. This is known as reflexive.

The notation R_L means an equivalence relation 'R' over the language L. The notation R_M means an equivalence relation R over machine M. We know for every regular language L there is a machine M that exactly accepts the strings in L.

Our R_L is defined xR_LY ⇔ for all z in Σ*
(xz in L ⇔ yz in L)

Our R_M is defined xR_MY ⇔ xzR_Myz for all z in Σ*

In other words,

$\delta(q0, xz) = \delta(\delta(q0,x),z))$
$= \delta(\delta(q0,y),z))$
$= \delta(q0, yz)$

For x,y, z strings in Σ*.

R_M divides the set Σ* into equivalence classes, one class for each state reachable in that particular state of M from the starting state q0. To get R_L from this we have to consider only the final reachable states of M.

Statement of Myhill-Nerode Thoerem:

The Myhill-Nerode theorem states the following three statements are equivalent:

1. The set L, a subset of Σ*, is accepted by a DFA. (We know this means L is regular language).
2. L is the union of some of the equivalence classes of a right invariant (with respect to concatenation) equivalence relation of finite index.
3. Let equivalence relation R_L be defined by: xR_Ly if and only if for all z in Σ*, xz is in L exactly when yz is in L. Then R_L is of finite index.

Implementation of Myhill-Nerode Theorem

Step 1: Let us start with a FA = (Q, Σ, δ, q0, F) as usual remove from Q, F and δ all states that cannot be reached from q0.

Step 2: Let us build a two dimensional matrix labeling the right side q0, q1, q2, … running down and denote this as the "p" first subscript. Label the top as q0, q1, q2, … and denote this as "q" second subscript.

Let us put dashes in the major diagonal and the lower triangular part of the matrix (every thing below the diagonal) we will always use the upper triangular part because $xR_My = yR_Mx$ is symmetric. We will also use (p, q) to index into the matrix with the subscript of the state called "p" always less than the subscript of the state called "q".

Step 3: We can have one of three things in a matrix location where there is no dash.
A "X" indicates a distinct state from our initialization, "x", at the location (p, q) indicate that state p is distinguishable from q.

We will level all empty matrix location with "0". The "0" location means the p and q are equivalent and will be the same state in the minimum machine.

Step 4: We begin for every pair of distinct states (p, q) in $F\times(Q-F)$ such that subscript of p < subscript of q. We do not write over dashes. If (p,q) has a dash, we put X in (q, p).

Step 5: We begin for every pair of distinct states (p, q) in F×F and for every pair of distinct state (p, q) in $(Q-F)\times(Q-F)$ we do

If for any input symbol, (r, s) has X or x then we put x in (p, q), we check (s, r) if (r, s) has a dash. (where $r = \delta(p, a)$ and $s = \delta(q, a)$ where $a \varepsilon \Sigma$.)

If (p, q) has a dash then we mark x at (q, p)

We do not have to write another "x" if one is there already.

Step 6: We mark all the empty position "0" in upper triangle.

Step 7: To find out the states in minimized DFA, we check all the rows of final matrix. The minimized machine will contain the states equal to the number of rows contains "0" entries (including the initial state if row starts with initial state does not contain any "0" entry).

If a row start with qx is having "0" at qy and qz positions then {qx, qy, qz} will a state in minimum machine.

Initial state will be that set which contains initial state as member state. (If there is no "0" entry at the row, starts with the initial state, then initial state of new minimized machine will be same as in old machine). Final states of minimized machine are all these sets which contains the any accepting state of old machine as member state.

Step 8: For finding the transition between the states of new machine, we will adopt the same strategy as we adopt in the minimization process discussed earlier, we start with the initial state of M and check transition of it for all input symbols in Σ. We repeat this process till we are getting any new state.

Example:

Minimize the given DFA by using Myhill-Nerode Theorem:

State	**a**	**b**
q0	q1	q4
q1	q2	q3
*q2	q7	q8
*q3	q8	q7
q4	q5	q6
*q5	q7	q8
*q6	q7	q8
q7	q7	q7
q8	q8	q8

Q = {q0,q1,q2,q3,q4,q5,q6,q7,q8}
Σ = {a,b}

Initial state = q0

F = {q2,q3,q5,q6}
Q − F = {q0,q1,q4,q7,q8}
We use an ordered F×F = {(q2,q3),(q2,q5),(q2,q6),(q3,q5),(q3,q6),(q5,q6)}
And (Q−F)×(Q−F) = {(q0,q1),(q0,q4),(q0,q7),(q0,q8),(q1,q4),(q1,q7), {(q1,q8),(q4,q7),(q4,q8),(q7,q8)}

Step 1: Now we build the matrix labeling the "p" rows q0,q1,q2, … and labeling the "q" columns q0,q1, … and we put the dashes (the symbol "–") **below the principal diagonal**

	q0	q1	q2	q3	q4	q5	q6	q7	q8
q0									
q1	–								
q2	–	–							
q3	–	–	–						
q4	–	–	–	–					
q5	–	–	–	–	–				
q6	–	–	–	–	–	–			
q7	–	–	–	–	–	–	–		
q8	–	–	–	–	–	–	–	–	

Step 2: Now we mark X at (p,q) in upper triangle such that p in F and q in (Q−F) as subscript of p is lower than the subscript of q or p in (Q−F) and q in F and q is lower than q.

F = {q2,q3,q5,q6}
Q − F = {q0,q1,q4,q7,q8}

F×(Q−F) or (Q−F)×F such that p is lower than q.
= {(q2,q4),(q2,q7),(q2,q8),(q3,q4),(q3,q7),(q3,q8),(q5,q7),(q5,q8),(q6,q8),(q6,q7),(q0,q2),(q0,q3),(q0,q5),(q0,q6),(q1,q2),(q1,q3),(q1,q5),(q1,q6),(q4,q5),(q4,q6)}

Now we mark × in the matrix given in following figure

	q0	**q1**	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**	**q8**
q0			X	X		X	X		
q1	–		X	X		X	X		
q2	–	–			X			X	X
q3	–	–	–		X			X	X
q4	–	–	–	–		X	X		
q5	–	–	–	–	–			X	X
q6	–	–	–	–	–	–		X	X
q7	–	–	–	–	–	–	–		
q8	–	–	–	–	–	–	–	–	

Step 3:

We mark "x" and "0" in the matrix:

First we consider Q−F

Q−F = {q0,q1,q4,q7,q8}, so ordered pair of
(Q−F) × (Q−F) = {(q0,q1),(q0,q4),(q0,q7),(q0,q8),(q1,q4),(q1,q7),{(q1,q8),(q4,q7),(q4,q8),(q7,q8)}

Now we will select (p,q) from (Q−F) × (Q−F) and
find (r_a,s_a) as $r_a = \delta(p,a)$ and $s_a = \delta(q,a)$ and
find (r_b,s_b) as $r_b = \delta(p,b)$ and $s_b = \delta(q,b)$

Now
If subscript of r is less than subscript of s we check (r,s)
If subscript of r is greater than subscript of s, the we check (s,r)
If at least of (r_a,s_a) [or (s_a,r_a)] and (r_b,s_b) [or (s_b,r_b)] is <u>either X or x</u>, then (p,q) will be <u>x</u>.
If both of (r_a,s_a) [or (s_a,r_a)] and (r_b,s_b) [or (s_b,r_b)] is <u>neither X nor x</u> then (p,q) will be "<u>0</u>".

i. For **(q0,q1)**,
$r_a = \delta(q0,a) = q1$ and $s_a = \delta(q1,a) = q2$, so next state pair (q1,q2)
$r_b = \delta(q0,b) = q4$ and $s_b = \delta(q1,b) = q3$, so next state pair (q4,q2) i.e. **(q2,q4)**

It can be checked from the matrix that (q1,q2) is X
so **(q0,q1)** will be x.

ii. For **(q0,q4)**,
$r_a = \delta(q0,a) = q1$ and $s_a = \delta(q4,a) = q5$, so next state pair (q1,q5)
$r_b = \delta(q0,b) = q4$ and $s_b = \delta(q4,b) = q6$, so next state pair (q4,q6)

It can be checked from the matrix that (q1,q5) is X
so **(q0,q4)** will be x.

iii. For **(q0,q7)**,
$r_a = \delta(q0,a) = q1$ and $s_a = \delta(q7,a) = q7$, so next state pair (q1,q7)
$r_b = \delta(q0,b) = q4$ and $s_b = \delta(q7,b) = q7$, so next state pair (q4,q7)

It can be checked from the matrix that (q1,q7) is neither X nor x
so **(q0,q7)** will be 0.

iv. For **(q0,q8)**,
$r_a = \delta(q0,a) = q1$ and $s_a = \delta(q8,a) = q8$, so next state pair (q1,q8)
$r_b = \delta(q0,b) = q4$ and $s_b = \delta(q8,b) = q8$, so next state pair (q4,q8)

It can be checked from the matrix that (q1,q8) is neither X nor x
so **(q0,q8)** will be x.

v. For **(q1,q4)**,
$r_a = \delta(q1,a) = q2$ and $s_b = \delta(q4,a) = q5$, so next state pair (q2,q5)
$r_b = \delta(q1,b) = q3$ and $s_b = \delta(q4,b) = q6$, so next state pair (q3,q6)

It can be checked from the matrix that (q2,q5) and (q3,q6) is neither X nor x
so **(q0,q1)** will be 0.

vi. For **(q1,q7)**,
$r_a = \delta(q1,a) = q2$ and $s_a = \delta(q7,a) = q7$, so next state pair (q2,q7)
$r_b = \delta(q1,b) = q3$ and $s_b = \delta(q7,b) = q7$, so next state pair (q3,q7)

It can be checked from the matrix that (q2,q7) is X
so **(q1,q7)** will be x.

vii. For **(q1,q8)**,
$r_a = \delta(q1,a) = q2$ and $s_a = \delta(q8,a) = q8$, so next state pair (q2,q8)
$r_b = \delta(q1,b) = q3$ and $s_b = \delta(q8,b) = q8$, so next state pair (q3,q8)

It can be checked from the matrix that (q2,q8) is X
so **(q1,q8)** will be x.

viii. For **(q4,q7)**,
$r_a = \delta(q4,a) = q5$ and $s_a = \delta(q7,a) = q7$, so next state pair (q5,q7)
$r_b = \delta(q4,b) = q6$ and $s_b = \delta(q7,b) = q7$, so next state pair (q6,q7)

It can be checked from the matrix that (q5,q7) is X
so **(q4,q7)** will be x.

ix. For **(q4,q8)**,
$r_a = \delta(q4,a) = q5$ and $s_a = \delta(q8,a) = q8$, so next state pair (q5,q8)
$r_b = \delta(q4,b) = q6$ and $s_b = \delta(q8,b) = q8$, so next state pair (q6,q8)

It can be checked from the matrix that (q5,q8) is X
so **(q4,q8)** will be x.

x. For **(q7,q8)**,
$r_a = \delta(q7,a) = q7$ and $s_a = \delta(q8,a) = q8$, so next state pair (q7,q8)
$r_b = \delta(q7,b) = q7$ and $s_b = \delta(q8,b) = q8$, so next state pair (q7,q8)

It can be checked from the matrix that (q7,q8) is neither X nor x
so **(q7,q8)** will be 0.

Now we consider F
F = {q2,q3,q5,q6}
so ordered pair of
F×F = {(q2,q3),(q2,q5),(q2,q6),(q3,q5),(q3,q6),(q5,q6)}

xi. For **(q2,q3)**,
$r_a = \delta(q2,a) = q7$ and $s_a = \delta(q3,a) = q8$, so next state pair (q7,q8)
$r_b = \delta(q2,b) = q8$ and $s_b = \delta(q3,b) = q7$, so next state pair (q8,q7) **i.e. (q7,q8)**

It can be checked from the matrix that (q7,q8) is neither X nor x so **(q2,q3)** will be 0.

xii. For **(q2,q5)**,
$r_a = \delta(q2,a) = q7$ and $s_b = \delta(q5,a) = q7$ so next state pair (q7,q7)
$r_b = \delta(q2,b) = q8$ and $s_b = \delta(q5,b) = q8$, so next state pair (q8,q8)

It can be checked from the matrix that both (q7,q7) and (q8,q8) is neither X nor x so **(q2,q5)** will be 0.

xiii. For **(q2,q6)**,
$r_a = \delta(q2,a) = q7$ and $s_b = \delta(q6,a) = q7$ so next state pair (q7,q7)
$r_b = \delta(q2,b) = q8$ and $s_b = \delta(q5,b) = q8$, so next state pair (q8,q8)

It can be checked from the matrix that (q7,q7) is neither X nor x and (q8,q8) is neither X nor x so **(q2,q6)** will be "0".

xiv. For **(q3,q5)**,
$r_a = \delta(q3,a) = q8$ and $s_b = \delta(q5,a) = q7$ so next state pair (q8,q7) **i.e. (q7,q8)**
$r_b = \delta(q3,b) = q7$ and $s_b = \delta(q5,b) = q8$, so next state pair (q7,q8)

It can be checked from the matrix that it is checked (q7,q8), it is neither X nor x so **(q3,q5)** will be "0".

xv. For **(q3,q6)**,
$r_a = \delta(q3,a) = q8$ and $s_b = \delta(q6,a) = q7$ so next state pair (q8,q7) **i.e. (q7,q8)**
$r_b = \delta(q3,b) = q7$ and $s_b = \delta(q6,b) = q8$, so next state pair (q7,q8)

It can be checked from the matrix that it is checked (q7,q8), it is neither X nor x so **(q3,q6)** will be "0".

xvi. For **(q5,q6)**,
$r_a = \delta(q5,a) = q7$ and $s_b = \delta(q6,a) = q7$ so next state pair (q7,q7)
$r_b = \delta(q5,b) = q8$ and $s_b = \delta(q6,b) = q8$, so next state pair (q8,q8)

It can be checked from the matrix that both of (q7,q7) and (q8,q8) are neither X nor x so **(q5,q6)** is "0".

	q0	**q1**	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**	**q8**
q0		x	X	X	x	X	X	0	0
q1	–		X	X	0	X	X	x	x
q2	–	–		0	X	0	0	X	X
q3	–	–	–		X	0	0	X	X
q4	–	–	–	–		X	X	x	x
q5	–	–	–	–	–		0	X	X
q6	–	–	–	–	–	–		X	X
q7	–	–	–	–	–	–	–		0
q8	–	–	–	–	–	–	–	–	

We are repeating Step 3 and we can find **new marking** for the following **two cases only**:

iii. For **(q0,q7)**,
$r_a = \delta(q0,a) = q1$ and $s_a = \delta(q7,a) = q7$, so next state pair (q1,q7)
$r_b = \delta(q0,b) = q4$ and $s_b = \delta(q7,b) = q7$, so next state pair (q4,q7)

It can be checked from the matrix that (q1,q7) is now x
so **(q0,q7) will be x instead of 0**.

iv. For **(q0,q8)**,
$r_a = \delta(q0,a) = q1$ and $s_a = \delta(q8,a) = q8$, so next state pair (q1,q8)
$r_b = \delta(q0,b) = q4$ and $s_b = \delta(q8,b) = q8$, so next state pair (q4,q8)

It can be checked from the matrix that (q1,q8) is now x
so **(q0,q8) will be x instead of 0**.

	q0	**q1**	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**	**q8**
q0		x	X	X	x	X	X	x	x
q1	–		X	X	0	X	X	x	x
q2	–	–		0	X	0	0	X	X
q3	–	–	–		X	0	0	X	X
q4	–	–	–	–		X	X	x	x
q5	–	–	–	–	–		0	X	X
q6	–	–	–	–	–	–		X	X
q7	–	–	–	–	–	–	–		0
q8	–	–	–	–	–	–	–	–	

Step 4:

The "0" at (x,y) means (x,y) belongs to a same class, so they make a single state.

So if resulting minimum machine is $M' = (Q, \Sigma, \delta', q0', F')$

$Q' = \{\{q0\}, \{q1,q4\}, \{q2,q3,q5,q6\}, \{q7,q8\}\}$
$F' = \{\{q2,q3,q5,q6\}\}$
$q0' = q0$

Replacing q0 by A
Replacing q1 and q4 by B
Replacing q2, q3, q5 and q6 by C
Replacing q7 and q8 by D,

We get

	a	**b**
A	B	B
B	C	C
*C	D	D
*C	D	D
B	C	C
*C	D	D
*C	D	D
D	D	D
D	D	D

In this particular transition table, a number of rows are redundant.

So the minimized DFA is as follows:

	a	**b**
A	B	B
B	C	C
*C	D	D
D	D	D

MODULE 2: FINITE AUTOMATA WITH OUTPUT

LECTURE 1: MOORE MACHINE AND MEALY MACHINE

1. MOORE MACHINE AND MEALY MACHINE

1.1. MOORE MACHINE

Moore Machine can be represented as a 6-tuple structure:

$M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$

where,

Q is set of states of the system
Σ is input alphabet of the system
Δ is output alphabet of the system
δ is transition function of the system defined as $\delta : Q \times \Sigma \rightarrow Q$
λ is output function of the system defined as $\lambda : Q \rightarrow \Delta$
q_0 is the initial or start state of the system

A Moore machine can be denoted by transition diagram or by transition table or by transition function.

Let us take an example of a Moore machine

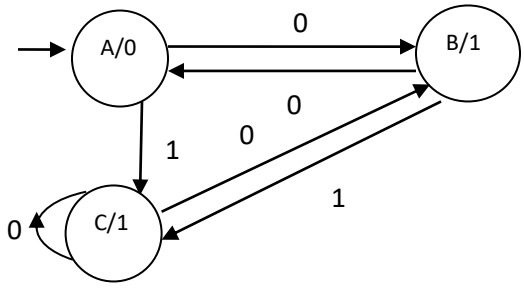


Figure 2.1: Transition Diagram of a Moore machine M1

We can represent the above Moore machine by a transition table as follows:

PS	NS		Output
	x=0	x=1	z
→ A	B	C	0
B	A	C	1
C	B	C	1

We can represent Moore machine M1 by 6-tuple structure as follows:

$Q = \{A,B,C\}$
$\Sigma = \{0,1\}$
$\Delta = \{0,1\}$
$\delta(A,0)=B$ $\delta(A,1)=C$ $\delta(B,0)=A$ $\delta(B,1)=C$ $\delta(C,0)=B$ $\delta(C,1)=C$
$\lambda(A) = 0$ $\lambda(B) = 1$ $\lambda(C) = 0$
$q_0 = A$

Processing an entire string by Moore machine:

For the input string 010, the transition of states is given by

$$A \rightarrow B \rightarrow C \rightarrow B$$

The output string is 0101.

For the input string ε, the output is $\lambda(A) = 0$

1.2. MEALY MACHINE

Mealy Machine can be represented as a 6-tuple structure:

$M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$

where,

Q is set of states of the system
Σ is input alphabet of the system
Δ is output alphabet of the system
δ is transition function of the system defined as $\delta : Q \times \Sigma \rightarrow Q$
λ is output function of the system defined as $\lambda : Q \times \Sigma \rightarrow \Delta$
q_0 is the initial or start state of the system

A Moore machine can be denoted by transition diagram or by transition table or by transition function.

Let us take an example of a Mealy machine

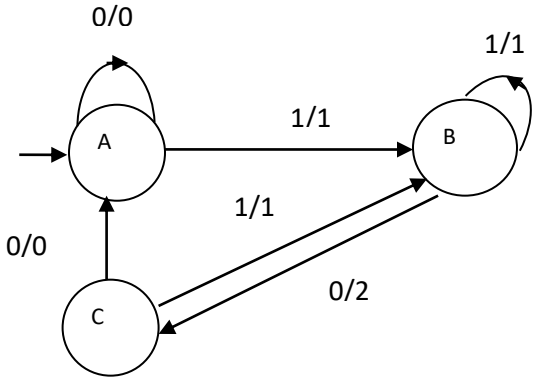


Figure 2.1: Transition Diagram of a Mealy machine M2

We can represent the above Mealy machine by a transition table as follows:

PS	NS, z	
	x=0	**x=1**
A	A,0	B,1
B	C,2	B,1
C	A,0	B,1

We can represent Mealy machine M2 by 6-tuple structure as follows:

$Q = \{A,B,C\}$
$\Sigma = \{0,1\}$
$\Delta = \{0,1,2\}$

$\delta(A,0)=A$	$\delta(A,1)=B$	$\delta(B,0)=C$	$\delta(B,1)=B$	$\delta(C,0)=A$	$\delta(C,1)=B$
$\lambda(A,0)=0$	$\lambda(A,1)=1$	$\lambda(B,0)=2$	$\lambda(B,1)=1$	$\lambda(C,0)=0$	$\lambda(C,1)=1$

$q_0 = A$

Processing an input string by Mealy machine:

For the input string 101, the transition of states is given by

$A \rightarrow B \rightarrow C \rightarrow B$

The output string is 121

For the input string ε, the output is ε

REMARKS:

- A finite automata can be converted into a Moore Machine by introducing $\Delta = \{0,1\}$ and defining $\lambda(q) = 1$ if and only if $q \in F$ and $\lambda(q) = 0$ if q does not belong to F.

- In case of a Moore machine if the input string is of length n, the output string is of length n+1; whereas in case of a Mealy machine if the input string is of length n, the output string is of also length n.

2. EQUIVALENCE OF MOORE MACHINE AND MEALY MACHINE

2.1. CONVERSION - MOORE MACHINE TO MEALY MACHIN

Let $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ be a given Moore machine, then $M = (Q, \Sigma, \Delta, \delta, \lambda', q_0)$ is Mealy machine equivalent to M where $\lambda'(q,a) = \lambda(\delta(q,a))$

Example:

Let a Moore machine is given as follows:

PS	NS		Output
	x=0	x=1	z
A	A	B	0
B	C	B	1
C	A	B	2

Now to get equivalent Mealy machine, we proceed as follows:

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

The equivalent Mealy machine is given as follows:

PS	NS, z	
	x=0	x=1
A	A,0	B,1
B	C,2	B,1
C	A,0	B,1

2.2. CONVERSION - MEALY MACHINE TO MOORE MACHINE

Let us consider the following Mealy machine.

PS	NS, z	
	x=0	x=1
q_1	q_3,0	q_2,0
q_2	q_1,1	q_4,0
q_3	q_3,1	q_1,1
q_4	q_4,1	q_3,0

We have to construct a Moore machine equivalent to that Mealy machine.

We look into the next state column for any state, say q_i, and determine the number of different outputs associated with q_i in that column.

We split q_i into several different states, the number of such states being equal to the number of different outputs associated with qi. For example, in this problem, q_1 is associated with one output 1 and q_2 is associated with one output 0. But, q_3 is associated with two outputs 0 & 1 and q_4 is associated with two outputs 0 & 1. So we split q_3 into q_{30} and q_{31}. Similarly, q_4 is split into q_{40} and q_{41}. Now table given in the example can be reconstructed for the new states as given by following table.

PS	NS, z	
	x=0	x=1
q_1	q_{30},0	q_2,0
q_2	q_1,1	q_{40},0
q_{30}	q_{31},1	q_1,1
q_{31}	q_{31},1	q_1,1
q_{40}	q_{41},1	q_{30},0
q_{41}	q_{41},1	q_{30},0

The pair of states and outputs which gives the Moore machine can be rearranged as given by following table.

PS	NS		z
	x=0	x=1	
q_1	q_{30}	q_2	1
q_2	q_1	q_{40}	0
q_{30}	q_{31}	q_1	0
q_{31}	q_{31}	q_1	1
q_{40}	q_{41}	q_{30}	0
q_{41}	q_{41}	q_{30}	1

LECTURE 2: SIMPLIFICATION OF MEALY MACHINE

3. SIMPLIFICATION OF MEALY MACHINE

3.1. DISTINGUISHABLE STATES AND EQUIVALENT STATES OF A (MEALY) MACHINE

Definitions of Distinguishable States and Equivalent States of a Machine:

Two states q_1 and q_2 of machine M are distinguishable if and only if there exists at least one finite input sequence, when applied to M, causes different output sequences, depending on whether q_1 and q_2 is the initial state. The input sequence which distinguishes these states is called a distinguishable sequence of the pair (q_1 and q_2). **If there exists for pair (q_1,q_2) a distinguishing sequence of length k, the states in (q_1,q_2) are said to be k-distinguishable.**

As an example let us consider the pair (A,B) of machine M3, whose state table is shown in following table.

PS	NS, z	
	x=0	x=1
A	E,0	D,1
B	F,0	D,0
C	E,0	B,1
D	F,0	B,0
E	C,0	F,1
F	B,0	C,0

The pair (A,B) is 1- distinguishable, since an input 1 applied to M3 when initially in A yields an output 1, versus an output 0 when it is initially in B. On the other hand, the pair (A,E) is 3-distinguishable, since there is no input sequence of length 2 which distinguishes A from E. The only sequence of length 3 which is a distinguishing sequence for the pair (A,E) is X = 111, and the output sequences corresponding to initial states A and E are 100 and 101, respectively. [We can note that 1101 is also a sequence which distinguishes A from E, although it is not the shortest such sequence. An all-zero sequence, on the other hand, will produce identical output sequences independently of whether the initial state is A or E.]

The concept of k-distinguishability leads directly to the definition of k-equivalence and equivalence. **States that are not k-distinguishable are said to be k-equivalent.** For example, states A and E of M3 are 2-equivanent (but they are not 3-equivalent). States which are k-equivalent are also r-equivalent, for all r < k. states that are k-equivalent for all k are said to be equivalent. Thus we arrive at the following definition.

Two States q_1 and q_2 of machine M are said to be <u>equivalent</u> if and only if, for every possible input sequence, the same (i.e. identical) output sequence will be produced regardless of whether q_1 and q_2 is the initial state. [This is normal Definition of equivalent states of a machine.**]**

Definition of equivalence can be generalized to the case where q_1 is a possible initial state of in machine M1, while q_2 is an initial state in machine M2, where both M1 and M2 have the same input alphabet.

If q_1 and q_2 are equivalent states, their corresponding x-successors for all x are either the same or also equivalent.

Hence, we can say that **two states q_1 and q_2 of machine M are said to be <u>equivalent</u> if and only if they produce the same (i.e. identical) output for every possible input x, and their x-successors, for all possible x, are either the same or also equivalent.** [This is recursive Definition of equivalent states of a machine.**]**

<u>Procedure of finding Distinguishable States and Equivalent States of a Machine:</u>

State equivalence is an equivalence relation and in consequence of this characteristic, the set of states of the machine can be partitioned into disjoint subsets, known as equivalence classes, so that two states are in the same equivalence class if and only if they are equivalent, and are in different classes if and only if they are distinguishable.

The first step is to partition the states of machine M into subsets such that all states in the same subset are 1-equivalent. This is accomplished by placing states having identical outputs under all possible inputs in the same subset. Clearly, two states which are in different subsets are 1-distinguishable.

As an example, we consider machine M3 given in table. The first partition P_0 corresponds to 0-distinguishability, and it defines our initial "ignorance" regarding the response of the various states, prior to the application of any input. P_1 is obtained simply by inspecting the table and placing those states having the same outputs, under all inputs, in the same block. Thus A,C,E are in the same block, since their outputs under inputs 0 and 1 are 0 and 1, respectively. A similar argument places B,D,F in the other block. Clearly, P_1 establishes the subsets (or classes or blocks) where states which are in a same or common subset are 1-equivalent but states which are in two different subsets are 1-distinguishable.

The next step is to obtain the partition P_2 where states which are in a same or common subset are 2-equivalent but states which are in two different subsets are 2-distinguishable. This is accomplished by observing that **two states are 2-equivalent if and only if they are 1-equivalent and their x-successors, for all possible x, are also same or 1-equivalent**. Consequently, two states are placed in the same (or common) block of P_2 if and only if they are in the same (or common) block of P_1, as well as, for each possible x, their x-successors are also contained in a same (or common) block of P_1. This step is carried out by splitting blocks of P_1 whenever their successors are **NOT** contained in a COMMON block of P_1. The 0-successor of

(A,C,E) is (C,E) and 1- successors of (A,C,E) is (B,D,F), and since both (C,E) is contained within a common block of P_1 as well as (B,D,F) is contained within a common block of P1, the states in the subset (A,C,E) are 2-equivalent; and therefore (A,C,E) constitutes a block or subset in P_2. The 1-successor of (B,D,F) is (D,B,C); but since (D,B) and (C) are NOT contained in a single block of P1, the block (B,D,F) must be split into (B,D) and (F) to obtain P_2.

In general, the P_{k+1} partition is obtained from P_k by placing in the same (or common) block of P_{k+1} those states which are in the same (or common) block of P_k, as well as, for each possible x, their x-successors are also contained in a same (or common) block of P_k.

Two states are <u>(k+1)-equivalent</u> if and only if they are k-equivalent and their x-successors, for all possible x, are also same or k-equivalent.

If for some k, $P_{k+1} = P_k$ the process terminates and P_k defines the sets of equivalent states of the machine; that is, all states contained in a same block of P_k are equivalent, where states belonging to different blocks are distinguishable. P_k is thus called the equivalence partition, and the foregoing procedure is referred to as Moore reduction procedure. For machine M3, P_3 is equivalence partition, and therefore states A & C are equivalent, and so are states B & D.

P_0 = {A,B,C,D,E,F}
P_1 = {{A,C,E},{B,D,F}}
P_2 = {{A,C,E},{B,D},{F}}
P_3 = {{A,C},{E},{B,D},{F}}
P_4 = {{A,C},{E},{B,D},{F}}

Equivalence partition for machine M3

3.2. Minimization of (Mealy) Machine

If we denote the blocks of the equivalence partition P_3 of M3 by α, β, γ and δ, respectively, to (A,C), (E), (B,D) and (F), we obtain the minimized machine M1′

PS	NS, z	
	x=0	**x=1**
α	β,0	γ,1
β	α,0	δ,1
γ	δ,0	γ,0
δ	γ,0	α,0

Replacing α, β, γ and δ, respectively, by A, B, C and D, we get

PS	NS, z	
	x=0	**x=1**
A	B,0	C,1
B	A,0	D,1
C	D,0	C,0
D	C,0	A,0

LECTURE 3: SIMPLIFICATION INCOMPLETELY SPECIFIED MACHINE

4. SIMPLIFICATION OF INCOMPLETELY SPECIFIED MACHINES:

Definition of Compatible States of a Machine:

Two States q_1 and q_2 of machine M are said to be compatible if and only if, for every possible input sequence applicable to both q_1 and q_2, the same (i.e. identical) output sequence will be produced whenever both outputs are specified and regardless of whether q_1 and q_2 is the initial state. [This is normal Definition of compatible states of a machine.**]**

If q_1 and q_2 are compatible states, their corresponding x-successors for all x are either the same or also compatible.

Hence, we can say that **two states q_1 and q_2 of machine M are said to be compatible if and only if they produce outputs which are not conflicting (i.e. same or identical when specified) for every possible input x, and their x-successors, for all possible x, are either the same or also compatible.** [This is recursive Definition of compatible states of a machine.**]**

In general, three or more states, q_1, q_2, q_3, . . . , are compatible if and only if , for every possible applicable input sequence, no two conflicting output sequences will be produced, without regard as to which of the above states is the initial states. Thus a set of states (q_1, q_2, q_3, . . .) is called a compatible if all its members are compatible.

A compatible C_i is said to be larger than, or to cover, another compatible C_j if and only if every state contained in C_j is also contained in C_i. A compatible is maximal if it is not covered by any other compatible. (Let us note that a single state that is not compatible with any other state is a maximal compatible). Thus, if we find the set of all the maximal compatibles, that in effect is equivalent to finding all compatibles, since every subset of a compatible is also a compatible.

Compatibility relation is not an equivalence relation. It thus follows that a set of states is a compatible if and only if every pair of states in that set is compatible. For example states **q_1, q_2, q_3** of a machine M will form the compatible ($q_1q_2q_3$) if and only if (q_1q_2), (q_1q_3) and (q_2q_3) are compatibles. **While the equivalence partition consists of disjoint blocks, the subsets of compatibles may be overlapped**.

The merger graph:

It is desirable first to generate the entire set of compatibles, and then to select an appropriate subset, which will form the basis for a state reduction leading to a minimal machine.

Since a set of states is compatible if and only if every pair of states in that set is compatible, it is sufficient to consider only pair of states and to use them to generate the entire set. We shall refer to a compatible pair of states as a compatible pair. Let the x-successors of A and B are C and D, respectively; then (AB) implies (CD), or (CD) is said to be implied by (AB). Thus, if (AB) is a compatible pair, then (CD) is referred to as its implied pair. The merger graph presented subsequently serves as the major tool in the determination of the set of all compatibles.

The merger graph of an n-state machine M is an undirected graph defined as follows:

1. It consists of n vertices, each of which corresponds to a state of M.
2. For each pair of states (AB) in M whose next-states and output entries are not conflicting, an undirected arc is drawn between vertices A and B.
3. If for a pair of states (AB) the corresponding outputs under all inputs are not conflicting, but next-states are not the same, an interrupted arc is drawn between A and B, and the implied pairs are entered in the space.

Let us consider a machine M4.

PS	**NS, z**			
	I_1	**I_2**	**I_3**	**I_4**
A	–	C,1	E,1	B,1
B	E,0	–	–	–
C	F,0	F,1	–	–
D	–	–	B,1	–
E	–	F,0	A,0	D,1
F	C,0	–	B,0	C,1

Its merger graph is shown in figure below:

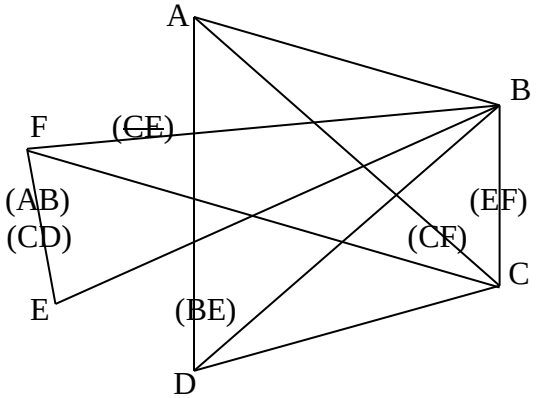


Figure 2.1a: Merger graph for machine M4

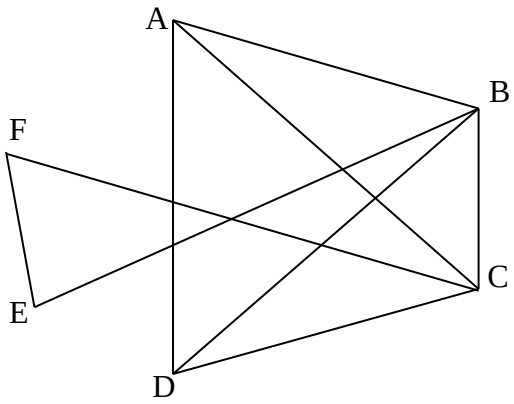


Figure 2.1b: Modified or final merger graph for machine M4

For machine M4 the merger graph reveals the existence of nine compatible pairs:

(AB),(AC),(AD),(BC),(BD),(BE),(CD),(CF),(EF)

Moreover, since (AB), (AC) and (BC) are compatibles, then (ABC) is also a compatible, and so on. In this manner the entire set of compatibles of M4 can be generated from its compatible pairs.

In order to find **a minimal set of compatibles**, which covers the original machine and can be used as a basis for the construction of a minimal machine, it is often useful to find the **set of maximal compatibles**. [A compatible is maximal if it is not contained in any other compatible.]

In terms of the merger graph, we are looking for **complete polygons** which are not contained within any higher-order complete polygons. [A complete polygon is one in which all possible diagonals exist.] Since the states covered by a complete polygon are all pair wise compatible, they constitute a compatible; and if the polygon is not contained in ant higher-order complete polygon, they constitute a maximal compatible.

In the figure of merger graph the set of highest-order polygons are the tetragon (ABCD) and the arcs (CF), (BE) and (EF). Thus the following set of maximal compatibles for machine M4 results:

{(ABCD),(BE),(CF),(EF)}

The closed sets of compatibles

A set of compatibles (for machine M) is said to be closed if, for every compatible contained in the set, all its implied compatibles are also contained in the set. A closed set of compatibles which contains all the states of M is called a **closed covering**.

Examples (for Machine M4):

{(BE),(CF),(EF)}	neither closed nor covering
{(AD),(BE),(CD)}	closed but not covering
{(ABCD),(EF)}	covering but not closed
{(ABCD),(BE),(CF),(EF)}	**closed and covering**
{(AD),(BE),(CF)} {(AB),(CD),(EF)}	**minimal closed covering**

Merger graph itself gives **set of maximal compatibles** which must implicitly be a **closed covering** but **not necessarily minimal**.

Now, it is desirable to look for a closed covering which yields a simpler machine.

Unfortunately, there is no simple, precise procedure leading to the selection of the minimal closed covering, and "trial-and-error" technique cannot be avoided

The preceding minimal closed coverings have been obtained by inspecting the merger graph and employing a "trail-and-error" procedure.

It should be pointed out that NO straightforward systematic procedure to find a minimal closed covering is known as yet, and a certain amount of search is unavoidable.

The compatibility graph

Let us consider the machine M5.

PS	**NS, z**			
	I_1	**I_2**	**I_3**	**I_4**
A	–	–	E,1	–
B	C,0	A,1	B,0	–
C	C,0	D,1	–	A,0
D	–	E,1	B,–	–
E	B,0	–	C,–	B,0

The merger graph is constructed in the usual manner. The set of maximal compatibles derived from the merger graph contains four members and is given by

{(ACD),(BC),(BE),(DE)}

A

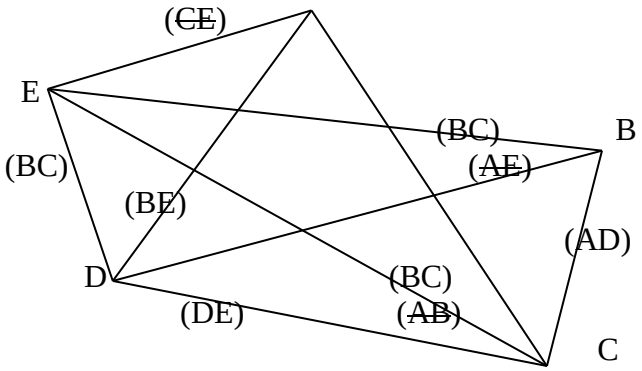


Figure 2.2a: Merger graph for machine M5

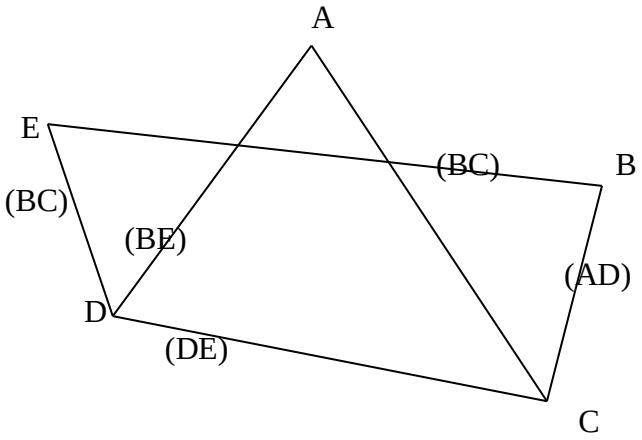


Figure 2.2b: Modified or final merger graph for machine M5

The compatibility graph is a directed graph whose vertices correspond to all compatible pairs, and an edge leads from vertex (AB) to vertex (CD) if and only if (AB) implies (CD). It is a tool which aids in the search for a minimal closed covering.

The compatibility graph for machine M4 is shown in following figure.

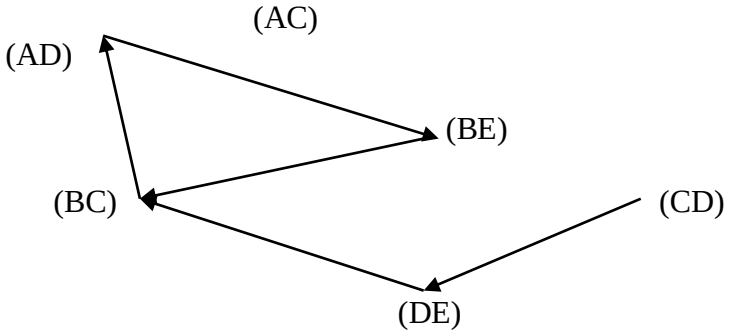


Figure 2.3: Compatibility graph for machine M5

A **subgraph** of a compatibility graph is said to be closed if, for every vertex in the subgraph, all outgoing edges and their terminating vertices also belong to the subgraph. If, in addition, every state of the machine is covered by at least one vertex of the subgraph, then the subgraph forms a closed covering for that machine.

Examples (for Machine M5):

{(AD),(BE)}	neither closed nor covering
{(AC)}	closed but not covering
{(AD),(BE),(CD)}	covering but not closed
{(DE), (BC), (AD), (BE)}	**closed and covering**
{(BC), (AD), (BE)}	**minimal closed covering**

The compatibility graph of above figure contains seven closed subgraphs [including (AC) alone and the graph itself], six of which form **closed coverings** for M5.

Compatibility graph itself gives a **closed covering** but **not necessarily minimal**.

Now, it is desirable to look for a closed covering which yields a simpler machine.

Unfortunately, there is no simple, precise procedure leading to the selection of the minimal closed covering, and "trial-and-error" technique cannot be avoided

The preceding minimal closed coverings have been obtained by inspecting the compatibility graph and employing a "trail-and-error" procedure.

It should be pointed out that NO straightforward systematic procedure to find a minimal closed covering is known as yet, and a certain amount of search is unavoidable.

[Note: If a closed subgraph containing the compatible pairs (AB), (BC), (AC) has been found, the compatible (ABC) can be formed, and so on.]

The triangle {(BC), (AD), (BE)} yields our desired **minimal closed covering** set, and hence, minimal machine which covers M5.

PS	**NS, z**			
	I_1	**I_2**	**I_3**	**I_4**
(AD) → α	–	γ, 1	γ, 1	–
(BC) → β	β,0	α,1	β/γ,0	α,0
(BE) → γ	β,0	α,1	β,0	β/γ,0

LECTURE 4: LOSSLESS AND LOSSY MACHINE

5. Lossless and Lossy machine:

A machine M is said to be (information) **lossless** if the knowledge of **the initial state, the output sequence and the final state** is sufficient to determine **uniquely** the **input sequence**.

A machine that is not lossless is called **lossy**. For example,

PS	**NS, z**	
	x=0	**x=1**
A	A,0	B,0
B	B,0	–

If the initial state is A, output sequence is 00 and the final state is B, the input sequence is either 01 or 10.

A machine is said to be (information) **lossless of finite order** if the knowledge of the **initial state and the first μ output symbols** is sufficient to determine **uniquely the first input symbol**. The integer **μ is called order of losslessness**, if μ is the **least** integer satisfying the above definition.

Test for Losslessness:

We have to test whether the following machine M6 is lossless or lossy.

PS	**NS, z**	
	x=0	**x=1**
A	A,1	C,1
B	E,0	B,1
C	D,0	A,0
D	C,0	B,0
E	B,1	A,0

We have to form Testing table for this machine. The upper part of the testing table is output-successor table. **For lower part of the table, if at least one at the next entry is blank, then blank will get priority, otherwise we have to form pair(s)**.

PS	NS	
	z=0	**z=1**
A	–	(AC)
B	E	B
C	(AD)	–
D	(BC)	–
E	A	B
AC	–	–
AD	–	–
BC	(AE)(DE)	
AE	–	(AB)(BC)
DE	(AB)(AC)	–
AB	–	(AB)(BC)

A machine is lossless if and only if its testing table does not contain any pair consisting of repeated entry.

As the testing table of this machine does not contain any pair consisting of repeated entry, this machine is lossless.

To find the order, we have to draw testing graph.

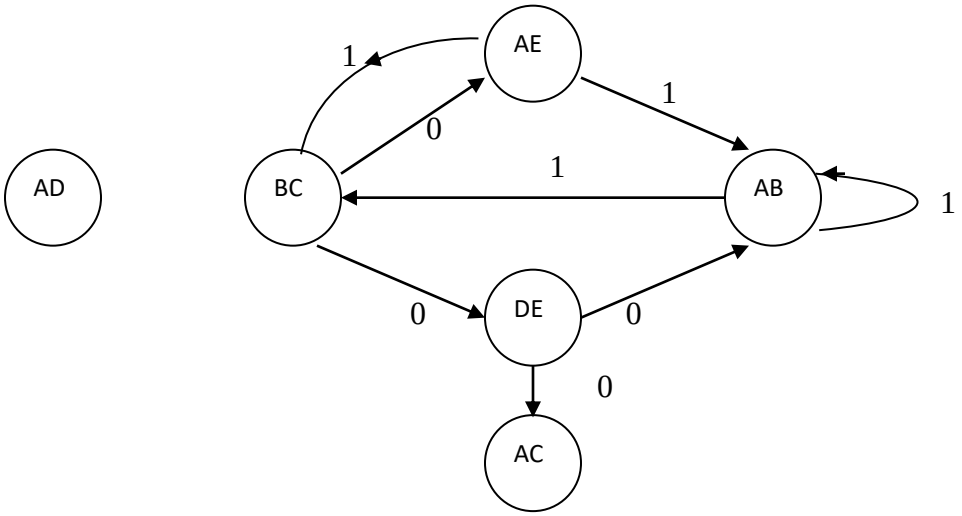


Figure 2.4: Testing Graph for machine M6

If testing graph contains at least one loop, then the machine is of infinite order.

As the testing graph of this machine contains loop, this machine is lossless of infinite order.

Theorem: A machine is lossless of finite order μ = l+2 if and only if its testing graph is loop-free and the length of the largest path in the graph is l.

Example:

Let us consider the following Machine M7.

PS	NS, z	
	x=0	x=1
A	A,0	B,0
B	C,0	D,0
C	D,1	C,1
D	B,1	A,1

Testing table:

PS	NS	
	z=0	z=1
A	(AB)	–
B	(CD)	–
C	–	(CD)
D	–	(AB)
AB	(AC)(AD)(BC)(BD)	–
CD	–	(AC)(AD)(BC)(BD)
AC	–	–
AD	–	–
BC	–	–
BD	–	–

Testing graph:

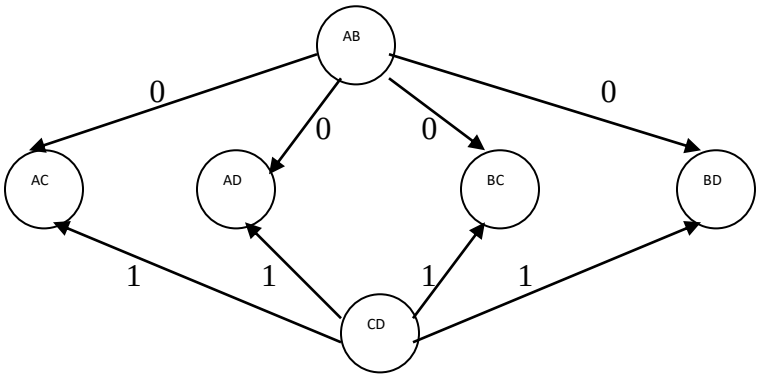


Figure 2.5: Testing Graph for machine M7

As l is 1, order μ = 1+2 = 3.

Retrieval of an input sequence (for a lossless machine):

Let us consider the following lossless machine:

PS	NS, z	
	x=0	**x=1**
A	A,1	C,1
B	E,0	B,1
C	D,0	A,0
D	C,0	B,0
E	B,1	A,0

Let us assume the **initial state is A**, **final state is B and output sequence is 110001100**. We have to find the corresponding **unique** input sequence.

Possible successors to initial state						A	A	A	A	
	A	A	A	B		D	B	B	D	B
		C	C	D	C	E	C	C	E	
Output sequence	1	1	0	0	0	1	1	0	0	
Possible predecessors to final state			C	B	C					
	A	A				A	A	C	D	B
			D	D	E					
The state sequence	A	A	C	D	C	A	A	C	D	B
Input sequence	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	

MODULE 3: REGULAR EXPRESSION

LECTURE 1: REGULAR EXPRESSION AND REGULAR SET AND REGULAR LANGUAGE

1. REGULAR EXPRESSION

The regular expressions are useful for representing certain sets of strings in an algebraic fashion**. Actually regular expressions describe the languages accepted by finite state automata**.

We give a formal recursive definition of regular expressions over Σ as follows:

- Any terminal symbol a (i.e., a Є Σ), ε andΦ are regular expression
- The **union** of two regular expressions R_1 and R_2, denoted by R_1+R_2 is also a regular expression.
- The **concatenation** of two regular expressions R_1 and R_2, denoted by R_1R_2 is also a regular expression.
- The **iteration (or closure)** of a regular expression R, denoted by R* is also a regular expression.
- If R is a regular expression, then (R) is also a regular expression.
- The regular expressions over Σ are precisely those obtained recursively by the application of the above rules once or several times.

2. REGULAR SET AND REGULAR LANGUAGE

Any set represented by a regular expression is called a regular set.

Examples are as follows:

Regular Expression	Regular Set
a	{a}
a+b	{a,b}
ab	{ab}
a*	{ε,a,aa,…}
(a+b)*	{ε,a,b,aa,ab,ba,bb,aaa,…}
ab+ba	{ab,ba}

For each regular expression r, we describe the language, it represents, which we denote as L(r). This is regular language. Actually, regular set and regular language are often used synonymously.

3. IDENTITIES AND ALGEBRAIC LAWS IN REGULAR EXPRESSION

Let, P,Q and R are three regular expressions. Then

i. $\Phi+R = R+\Phi = R$
ii. $\Phi R = R\Phi = \Phi$
iii. $\varepsilon R = R\varepsilon = R$
iv. $\varepsilon^* = \varepsilon$
v. $\Phi^* = \varepsilon$
vi. $R+R = R$
vii. $R^*R^* = R^*$
viii. $RR^* = R^*R = R^+$
ix. $(R^*)^* = R^*$
x. $\varepsilon+RR^* = \varepsilon+R^*R = \varepsilon+R^+ = R^*$

xi. $P+Q = Q+P$
xii. $PQ \neq QP$
xiii. $(PQ)^*P = P(QP)^*$
xiv. $(P+Q)^* = (P^*+Q)^* = (P+Q^*)^* = (P^*+Q^*)^* = (P^*Q^*)^* \neq (PQ^*)^* \neq (P^*Q)^*$
xv. $(P+Q)+R = P+(Q+R)$
xvi. $(PQ)R = P(QR)$
xvii. $P(Q+R) = PQ + PR$ and $(Q+R)P=QP+RP$
xviii. If $R = Q+RP$ then $R = QP^*$ (Arden's Theorem)

Proof of Arden's Theorem:

$R = Q + RP$
$= Q + (Q + RP) P$
$= Q + QP + RPP$
$= Q + QP + (Q + RP) PP$
$= Q + QP + QPP + RPPP$
$= Q + QP + QPP + QPPP + \ldots$
$= Q (\varepsilon + P + PP + PPP + \ldots)$
$= QP^*$ [proved]

LECTURE 2: REGULAR EXPRSSION TO FINITE AUTOMATA

4. CONSTRUCTION OF FINITE AUTOMATA FOR REGULAR EXPRSSION

If R is a regular expression over Σ representing regular language L, then there exists an NFA M with ε-transitions such that M accepts L

Let L(R) denotes the set or language represented by R.

Basis:

There are three parts of the basis, shown in Figure 3.1.

In part (a) we see how to handle the regular expression R which is ε. The language of the automation is easily seen to be L(R) = {ε}, since the only path from the start state to an accepting state is labeled ε.
In part (b) we can see the construction for regular expression R which is Φ. Clearly there are no paths from start state to accepting state, so L(R) = Φ is the language for this automation.
Finally, part (c) gives the automation for a regular expression R which is a where a Є Σ. The language L(R) of this automation evidently consists of the one string a, which is also {a}.

Figure 3.1 (a): NFA M1 which accepts the regular language L = {ε}

Figure 3.1 (b): NFA M2 which accepts the regular language L = Φ

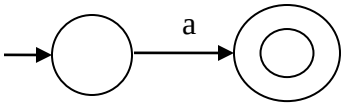


Figure 3.1 (c): NFA M3 which accepts the regular language L = {a}

Induction:

The three parts of the induction are shown in Figure 3.2.

Case i:
The expression is R_1+R_2 for some smaller expressions R_1 and R_2. Then the automation of Figure 3.2(a) serves. That is, starting at the new start state, we can go to the start state of either the automation for R_1 or the automation for R2 through ε-arc. We then reach the accepting state of one of these automata, following a path labeled by some string in $L(R_1)$ and $L(R_2)$, respectively. Once we reach the accepting state of the automation for R_1 or R_2, we can follow one of the ε-arcs to the accepting state of the new automation. Thus the language of the new automation in Figure 3.2(a) $L(R_1+R_2)$ is $L(R_1) \cup L(R_2)$.

Case ii:
The expression is R_1R_2 for some smaller expressions R_1 and R_2. The automation for the concatenation is shown in Figure 3.2(b). Let us note that the start state of the first automation becomes the start state of the whole, and the accepting state of the second automation becomes the accepting state of the whole. The idea is that the only paths from start to the accepting state go first through the automation for R_1, where it must follow a path labeled by a string in $L(R_1)$ and then through the automation for R_2, where it follows a path labeled by a string in $L(R_2)$. Thus, the paths in the automation of Figure 3.2(b) are all and only those labeled by strings in $L(R_1)L(R_2)$. That is, $L(R_1+R_2) = L(R1)\ L(R_2)$.

Case iii:
The expression is R* for some smaller expression R. Then we use the automation of Figure 3.2(c). That automation allows us to go either

- Directly from the start state to the accepting state along a path labeled ε. That path lets us accept ε, which is in L(R*) no matter what expression R is.
- To the start state of the automation for R, through that automation one or more times, and then to the accepting state. This set of paths allows us to accept strings in L(R), L(R)L(R), L(R)L(R)L(R), and so on, thus covering all strings in L(R*) except perhaps ε, which was covered by the direct arc to the accepting state mentioned previously.

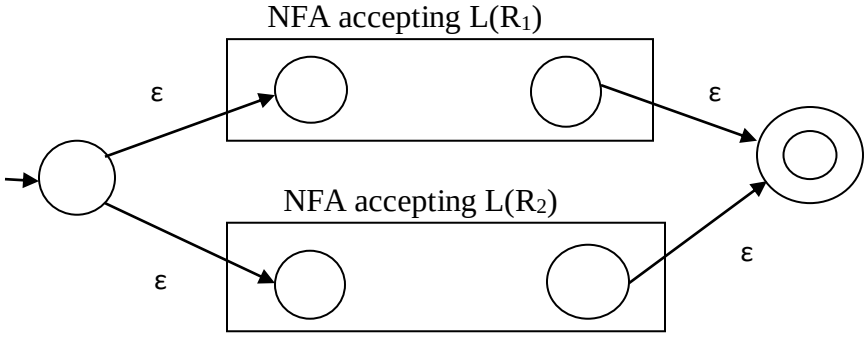


Figure 3.2 (a): NFA M4 which accepts the regular language $L(R_1+R_2)$

Figure 3.2 (b): NFA M5 which accepts the regular language $L(R_1R_2)$

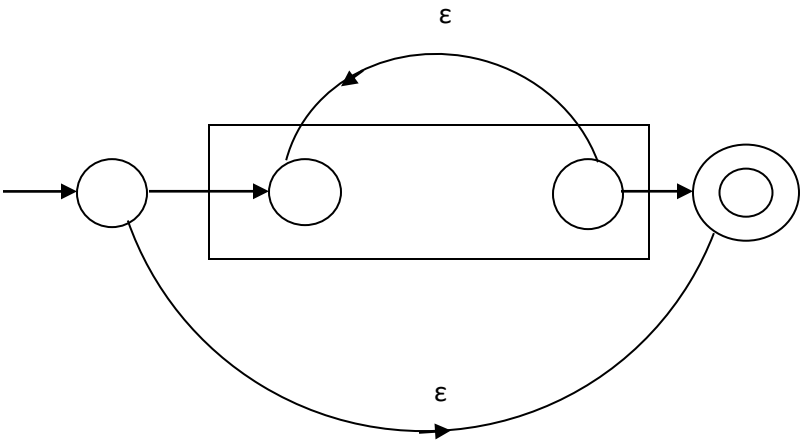


Figure 3.2 (c): NFA M6 which accepts the regular language $L(R_1^*)$

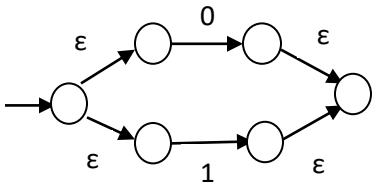
Example:

Let us convert the regular expression (0+1)*0 to an ε-NFA.

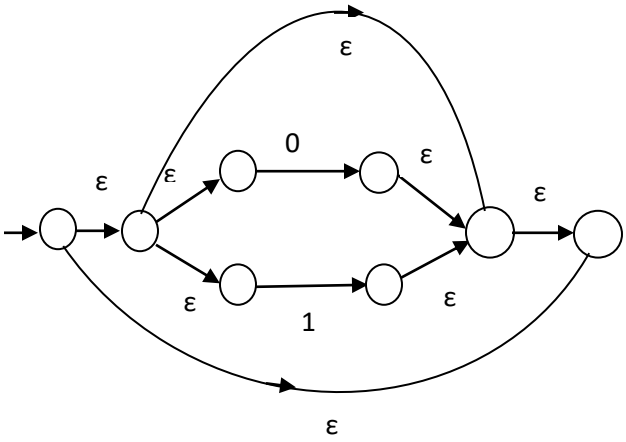
Our first step is to construct automation for 0+1. We use two automata constructed according to Figure 3.1(c), one with label 0 on the arc and one with label 1. These two automata are then combined using the union construction of Figure 3.2(a). n. The result is shown in Figure 3.3(a).

Next, we apply to Figure 3.3(a) the star construction of Figure 3.2(c). This automation is shown in Figure 3.3(b).

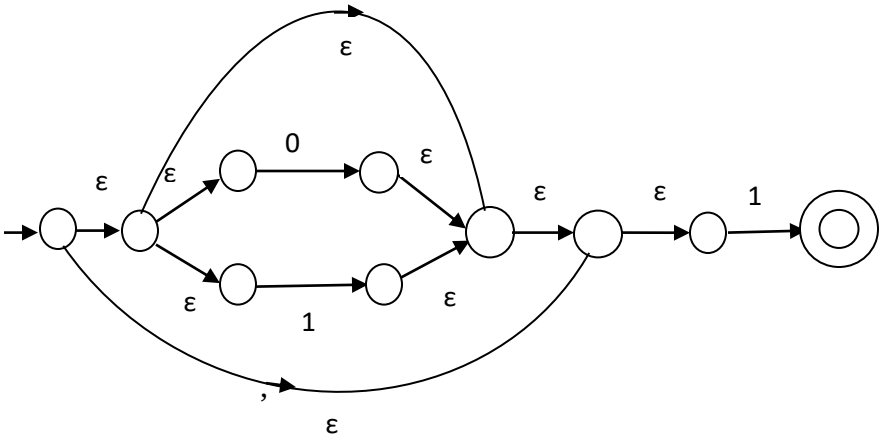
The last step involves applying the concatenation construction of Figure 3.2(b). We connect the automation of Figure 3.3(b) to another automation designed to accept only the string 0. This automation is another application of the basis construction of Figure 3.1(c) with label 0 on the arc. Let us note that we must create a new automation to recognize 0; we must not use the automation for 0 that was part of Figure 3.3(a). The complete automation is shown in Figure 3.3(c).



(a)



(b)



(c)

Figure 3.3: ε-NFA constructed for regular expression (0+1)*0

LECTURE 3: FINITE AUTOMATA TO REGULAR EXPRSSION

5. CONSTRUCTION OF REGULAR EXPRESSION FROM DFA

There are certain assumptions which are made regarding the transition system:

(i) The transition diagram should not have ε-transitions.
(ii) It must have only a single initial state.
(iii) It vertices are $q_1, q_2, \ldots, q_n$.
(iv) q_i is final state
(v) w_{ij} denotes the regular expression representing the set of labels of edges from q_i to q_j. We can get the following set of equation in $q_1, q_2, \ldots, q_n$.

$q_1 = q_1w_{11} + q_2w_{21} + \ldots + q_nw_{n1} + \varepsilon$ (ε is added since q_1 is the initial state)
$q_2 = q_1w_{12} + q_2w_{22} + \ldots + q_nw_{n2}$

…

$q_n = q_1w_{1n} + q_2w_{2n} + \ldots + q_nw_{nn}$

We solve these equation for q1 in terms of w_{ij}' s and it will be required regular expression. One thing should be noted that we add ε (empty string) in the equation starts with starting state q_1 and we solve equation to find out q_i (final state) in terms of w_{ij}' s, it is one of the regular expression for given deterministic finite automation**.**

Example:

Let us find the regular expression for the following DFA of Figure 3.4:

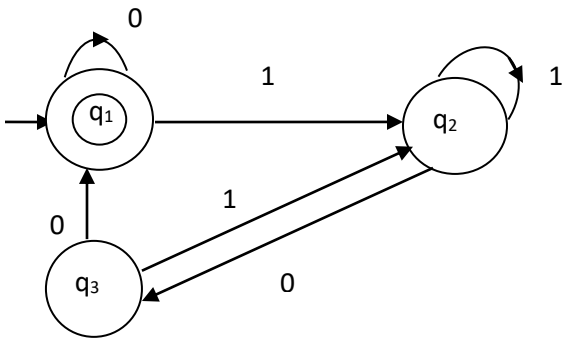


Figure 3.4: Transition Diagram of a DFA M1

Solution:

Let us form the equations:

$q_1 = q_10 + q_30 + \varepsilon$ … (1)
$q_2 = q_11 + q_21 + q_31$ … (2)
$q_3 = q_20$ … (3)

Now,

From (2),

$q_2 = q_11 + q_21 + q_31$
$= q_11 + q_21 + (q_20)1$ [From (3) replacing q_3 by q_20]
$= q_11 + q_21 + q_201$
$= q_11 + q_2(1+01)$
$= q_11(1+01)^*$ [Applying Arden's theorem which is if R = P+RP then R = QP*]

From (1),

$q_1 = q_10 + q_30 + \varepsilon$
$= q_10 + (q_20)0 + \varepsilon$ [From (3) replacing q_3 by q_20]
$= q_10 + (q_11(1+01)^*)00 + \varepsilon$
$= q_10 + q_11(1+01)^*00 + \varepsilon$
$= \varepsilon\ (0 + 1(1+01)^*00)^*$
$= (0 + 1(1+01)^*00)^*$

So regular expression is $(0 + 1(1+01)^*00)^*$

LECTURE 4: PROPERTIES OF REGULAR EXPRSSION

6. PUMPING LEMMA

Theorem

Let L be a regular language. Then there exists a constant **'c'** such that for every string **w** in **L** −
|w| ≥ 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**. We use the pumping lemma to obtain a contradiction −

- We select **w** such that **|w| ≥ c**
- We select **y** such that **|y| ≥ 1**
- We select **x** such that **|xy| ≤ c**
- We assign the remaining string to **z.**
- We select **k** such that the resulting string is not in **L.**

Hence L is not regular.

Problem

Prove that $\mathbf{L = \{a^ib^i \mid i \geq 0\}}$ 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 a's = $(p + 2q + r) = (p + q + r) + q = n + q$
- Hence, $xy^2z = a^{n+q}\, b^n$. Since $q \neq 0$, xy^2z is not of the form a^nb^n.
- Thus, xy^2z is not in L. Hence L is not regular.

7. CLOUSURE PROPERTIES

Regular languages are closed under the operation

i. Union
ii. Intersection
iii. Set Complement
iv. Set Difference
v. String Reversal
vi. Concatenation
vii. Star or Closure

Regular languages are NOT closed under the operation

i. Subset

EXAMPLES:

1. Design a Finite Automata (FA) that accepts set of all strings over Σ = {0,1} such that every string ends with 00 (i.e. every string ends with consecutive two 0's).

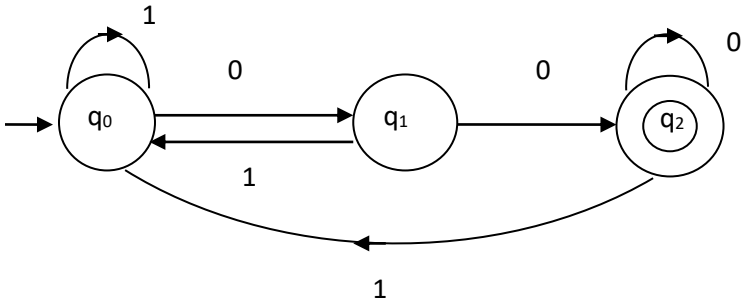


Figure 3.5: DFA for example 1

2. Design a Finite Automata (FA) that accepts set of all strings over Σ = {0,1} containing exactly one 0.

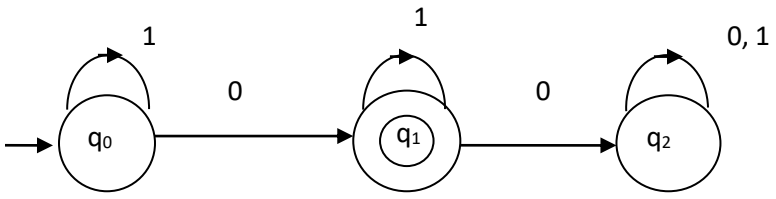


Figure 3.6: DFA for example 2

3. Design a Finite Automata (FA) that accepts set of all strings over Σ = {0,1} such that number of 0's is multiple of 3 (i.e. number of 0's is divisible by 3).

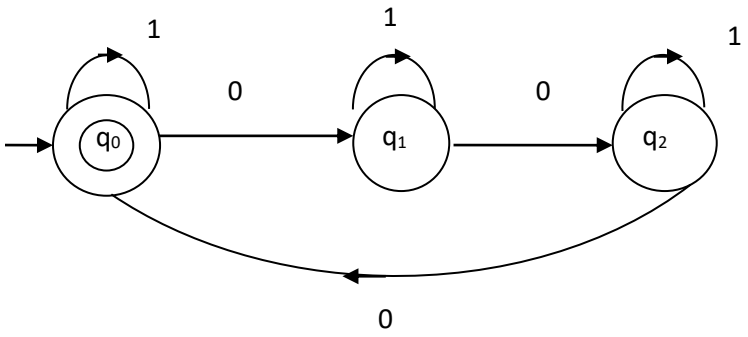


Figure 3.7: DFA for example 3

4. Design a Finite Automata (FA) M that accepts the language **L(M) = { w ∈ {a, b}* | w ends in the substring ab}**

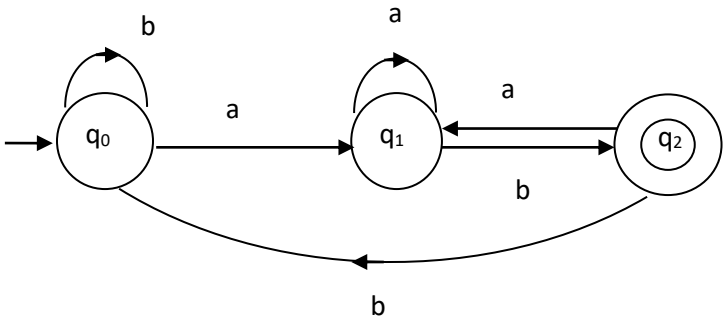


Figure 3.8: DFA for example 4

5. Design a Finite Automata (FA) M that accepts the language **L(M) = { 0^m1^n, m>=0,n<2}**

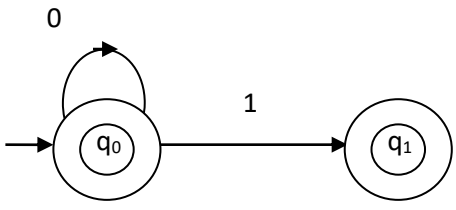


Figure 3.9: FA for example 5

6. Design a Finite Automata (FA) M such that that accepts the regular set S = {ab,ba} over the alphabet Σ = {a,b}.

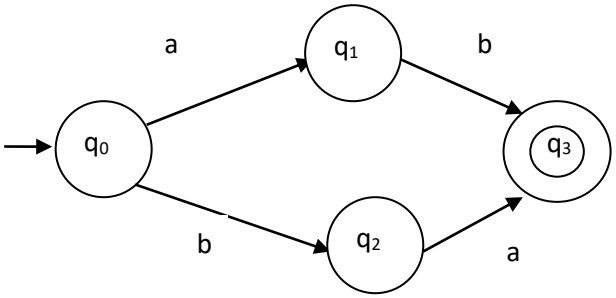


Figure 3.10: FA for example 6

7. Design a Finite Automata (FA) M such that that accepts the regular expression (0+1)*0 over Σ = {0,1}.

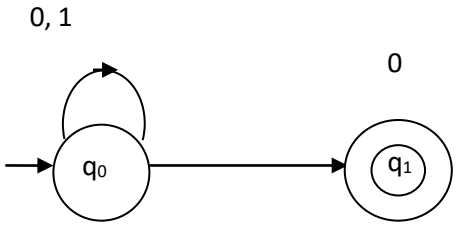


Figure 3.11: FA for example 7

8. Write the regular expression for the set of all strings over Σ = {0,1} such that every string ends with 00 (i.e. every string ends with consecutive two 0's).

   (0+1)*00

9. Write the regular expression for the set of all strings over Σ = {0,1} such that every string contains exactly one 0.

   1*01*

10. Write the regular expression for the set of all strings over Σ = {0,1} such that number of 0's is multiple of 3 (i.e. number of 0's is divisible by 3).

(1*01*01*0)*

11. Write the regular expression for the language **L = { w ∈ {a, b}* | w ends in the substring ab}**

    (a+b)*ab

12. Write the regular expression 'r' over Σ = {0,1} such that **L(r) = { 0^m1^n, m>3,n<4}**

    00000*(ε+1+11+111)

13. Write the regular expression for the set S **= {ab,ba}** over the alphabet Σ = {a,b}.

    ab+ba

14. Write the regular expression 'r' over Σ = {0,1} for the following FA

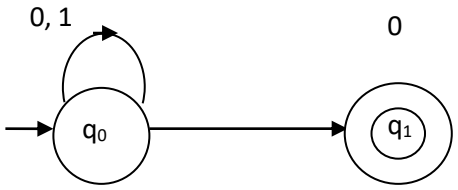


Figure 3.12

(0+1)*0

MODULE 4: GRAMMAR AND PUSH DOWN AUTOMATA

LECTURE 1: CONTEXT FREE GRAMMAR

DEFINITION:

A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

- **N** is a set of non-terminal symbols.
- **T** is a set of terminals where **N ∩ T =NULL**.
- **P** is a set of rules, **P: N → (N U T)***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
- **S** is the start symbol.

EXAMPLE:

1. The grammar ({A}, {a, b, c}, P, A), P : A → aA, A →abc.
2. The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S →ε
3. The grammar ({S, F}, {0, 1}, P, S), P: S → 00S|11F, F → 00F |ε

DERIVATION TREE:

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

REPRESENTATION TECHNIQUE:

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

EXAMPLE:

If S → x_1x_2 …… xn is a production rule in a CFG, then the parse tree / derivation tree will be as follows:

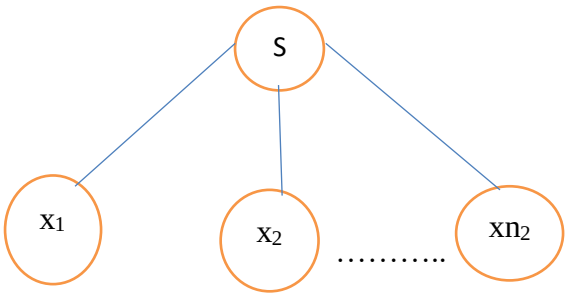
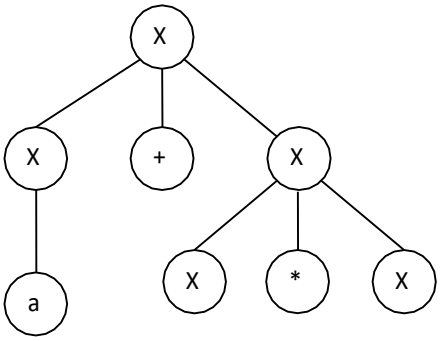
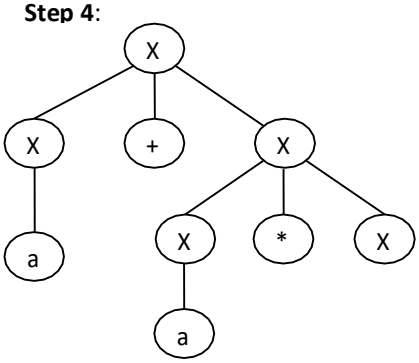


FIG3.1: Derivation Tree for the production S → x_1x_2 …… xn

Step 3:



Step 4:



DERIVATION OR YIELD OF A PARSE TREE:

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

Example

Let a CFG {N,T,P,S} be

N = {S}, T = {a, b}, Starting symbol = S, P = S → SS | aSb | ε

One derivation from the above CFG is "abaabb"

S → SS → aSbS →abS → abaSb → abaaSbb → abaabb

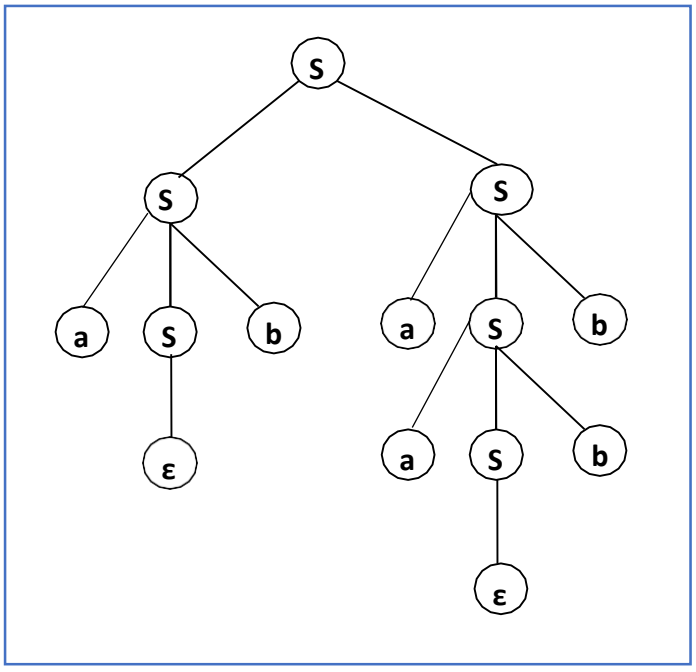


FIG 3. :

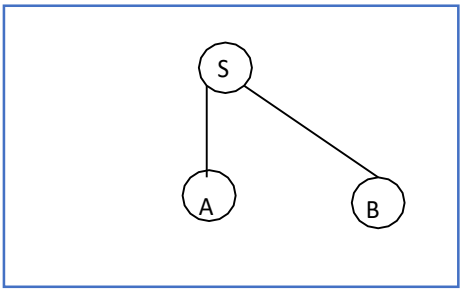
Sentential Form and Partial Derivation Tree

A partial derivation tree is a sub-tree of a derivation tree/parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree.

Example

If in any CFG the productions are:

S→AB, A → aaA | ε, B →Bb| ε the partial derivation tree can be the following:



If a partial derivation tree contains the root **S**, it is called a **sentential form**. The above sub-tree is also in sentential form.

Leftmost and Rightmost Derivation of a String

Leftmost derivation - A leftmost derivation is obtained by applying production to the leftmost variable in each step.

Rightmost derivation - A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Example

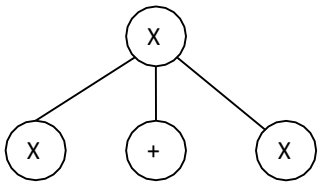
Let any set of production rules in a CFG be X → X+X | X*X |X| a
over an alphabet {a}.

The leftmost derivation for the string "**a+a*a**" may be:

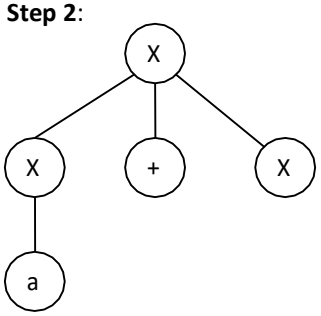
X → X+X→ a+X→ a+ X*X →a+a*X→a+a*a

The stepwise derivation of the above string is shown as below

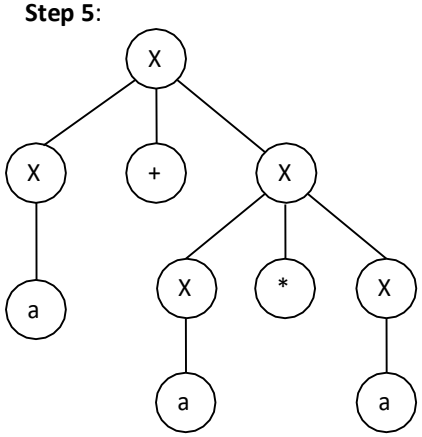
Step 1:



Step 2:



Step 5:

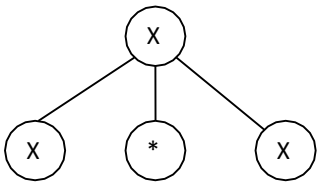


The right most derivation for the above string "**a+a*a**" may be:

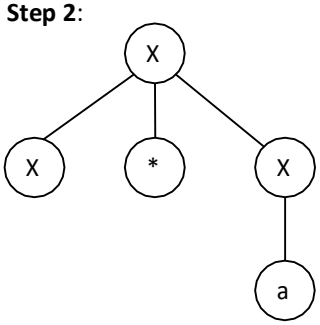
X → X*X→ X*a → X+X*a →X+a*a→a+a*a

The stepwise derivation of the above string is shown as below:

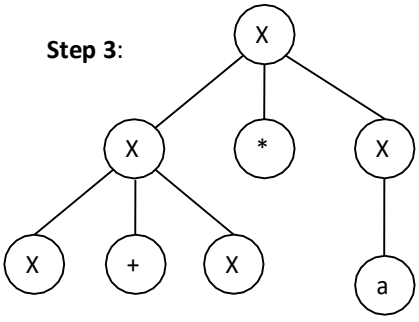
Step 1:



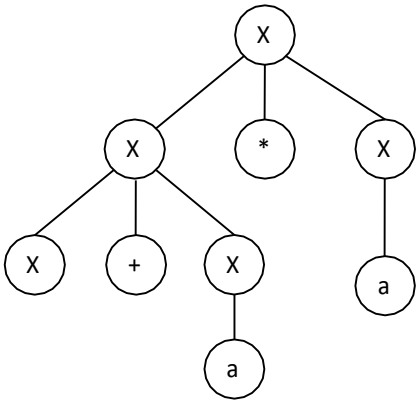
Step 2:



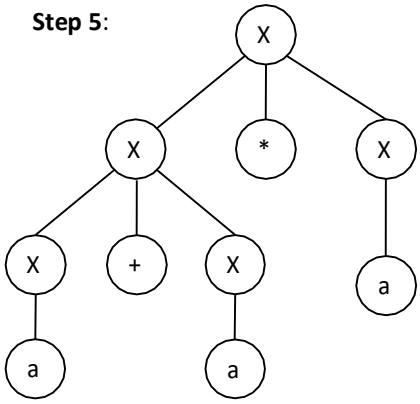
Step 3:



Step 4:



Step 5:



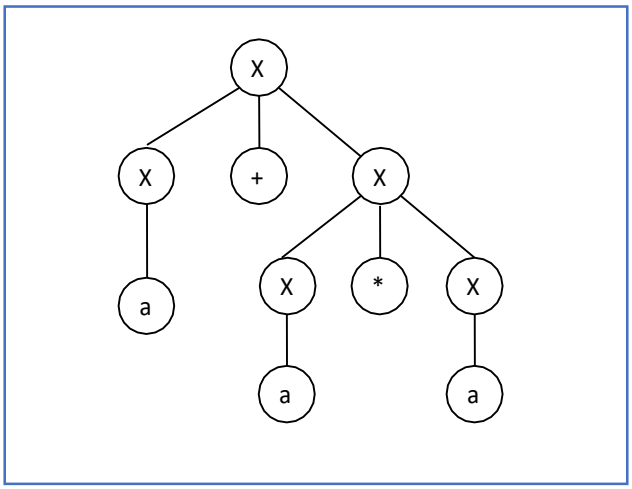
LEFT AND RIGHT RECURSIVE GRAMMAR:

In a context-free grammar G, if there is a production in the form X → Xa where X is a non-terminal and 'a' is a string of terminals, it is called a left recursive production. The grammar having a left recursive production is called a left recursive grammar.
And if in a context-free grammar G, if there is a production is in the form X → aX where X is a non-terminal and 'a' is a string of terminals, it is called a right recursive production. The grammar having a right recursive production is called a right recursive grammar.
IfacontextfreegrammarGhasmorethanonederivationtreeforsomestringw∈L(G), it is called an ambiguous grammar. There exist multiple right-most or left-most derivations for some string generated from that grammar.

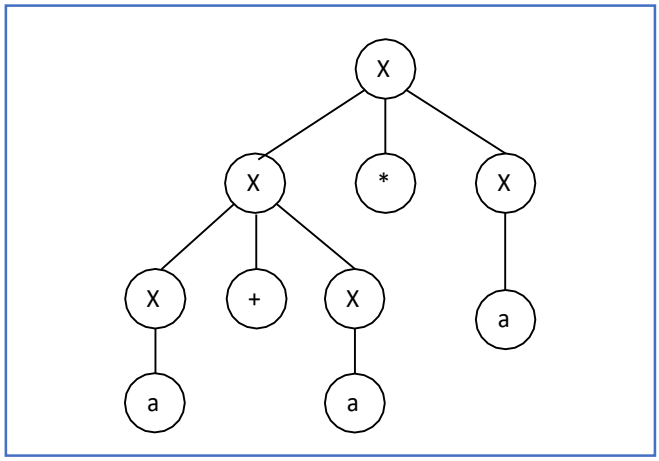
PROBLEM

Check whether the grammar **G** with production rules:
X → X+X | X*X |X| a is ambiguous or not.
Solution
Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

Derivation 1: X → X+X→ a +X→ a+ X*X →a+a*X→ a+a*a
PARSE TREE 1:



Derivation2: X → X*X→X+X*X→ a+ X*X →a+a*X→a+a*a
PARSE TREE 2:



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

LECTURE 2: MINIMIZATION OF CONTEXT FREE GRAMMAR

CFG SIMPLIFICATION:

In a CFG, it may happen that all the production rules and symbols are not needed for the derivation of strings. Besides, there may be some null productions and unit productions. Elimination of these productions and symbols is called **simplification of CFGs**. Simplification essentially comprises of the following steps:

- Reduction of CFG
- Removal of Unit Productions
- Removal of Null Productions

Reduction of CFG
CFGs are reduced in two phases:
Phase 1: Derivation of an equivalent grammar, **G'**, from the CFG, **G**, such that each variable derives some terminal string.
Derivation Procedure**:**
Step1: Include all symbols, $\mathbf{W_1}$, that derives one terminal and initialize **i=1**.
Step2: Include all symbols, $\mathbf{W_{i+1}}$, that derive $\mathbf{W_i}$.
Step3: Increment **i** and repeat Step 2, until $\mathbf{W_{i+1} = W_i}$.
Step4: Include all production rules that have $\mathbf{W_i}$ in it.

Phase 2: Derivation of an equivalent grammar, **G"**, from the CFG, **G'**, such that each symbol appears in a sentential form.
Derivation Procedure:
Step1: Include the start symbol in $\mathbf{Y_1}$and initialize **i =1**.
Step2: Include all symbols, $\mathbf{Y_{i+1}}$, that can be derived from $\mathbf{Y_i}$ and include all production rules that have been applied.
Step3: Increment **i** and repeat Step 2, until $\mathbf{Y_{i+1} = Y_i}$.
PROBLEM:
Find a reduced grammar equivalent to the grammar **G**, having production rules, P: S → AC | B, A → a, C → c | BC, E → aA |e
Solution
Phase 1:
T= { a, c, e}
W_1 = { A, C, E } from rules A → a, C → c and E → aA
W_2 = { A, C, E } U { S } from rule S → AC
W_3 = { A, C, E, S } U ϕ

Since $W_2 = W_3$, we can derive G' as:
G' = { { A, C, E, S }, { a, c, e }, P, {S}}
where P: S → AC, A → a, C → c , E → aA | e

Phase 2:
Y_1 = { S }
Y_2 = { S, A, C } from rule S → AC
Y_3 = { S, A, C, a, c } from rules A → a and C → c
Y_4 = { S, A, C, a, c }
Since $Y_3 = Y_4$, we can derive G" as:
G" = { { A, C, S }, { a, c }, P, {S}}
where P: S → AC, A → a, C → c
REMOVAL OF UNITPRODUCTIONS:
Any production rule in the form A → B where A, B ∈ Non-terminal is called **unit production**.

Removal Procedure:
Step1: To remove **A→B**, add production **A→x** to the grammar rule whenever **B→x** occurs in the grammar. [**x** ∈ Terminal, **x** can be Null]
Step2: Delete **A→B** from the grammar.
Step3: Repeat from step 1 until all unit productions are removed

PROBLEM:
Remove unit production from the following:
S → XY, X → a, Y → Z | b, Z → M, M → N, N → a

Solution**:**
There are 3 unit productions in the grammar:
Y→Z, Z→M, and M →N

At first, we will remove M → N.
As N→a, we add M →a, and M →Nis removed. The production set becomes
S → XY, X → a, Y → Z | b, Z → M, M → a, N → a

Now we will remove Z → M.
As M →a, we add Z→a, and Z→M is removed. The production set becomes
S → XY, X → a, Y → Z | b, Z → a, M → a, N → a
Now we will remove Y → Z.
As Z→a, we add Y→a, and Y→Z is removed. The production set becomes
S → XY, X → a, Y → a | b, Z → a, M → a, N → a
Now Z, M, and N are unreachable, hence we can remove those. The final CFG is unit production free:
S→XY, X→a, Y → a |b

Removal of Null Productions
In a CFG, a non-terminal symbol '**A**' is a nullable variable if there is a production **A** → **ϵ** or there is a derivation that starts at **A** and finally ends up with
ϵ: A → .....… →ϵ
Removal Procedure:

Step1 Find out nullable non-terminal variables which deriveϵ.
Step2 For each production A → a, construct all productions A → x where x is obtained from 'a' by removing one or multiple non-terminals from Step1.
Step3 Combine the original productions with the result of step 2 and remove ϵ- productions.

Problem
Remove null production from the following: S→ASA | aB | b, A → B, B → b | ϵ

Solution*:*
There are two nullable variables: A and B
At first, we will remove B → ϵ.
After removing B → ϵ, the production set becomes: S→ASA | aB | b | a, A → B| b | ϵ, B → b

Now we will remove A → ϵ.
After removing A → ϵ, the production set becomes:
S→ASA | aB | b | a | SA | AS|S, A →B|b, B → b This is the final production set without null transition.

LECTURE 3: NORMAL FORM OF CONTEXT FREE GRAMMARS

NORMAL FORMS:
There are two normal forms for Context Free Grammar, given as follows:

- CHOMSKY NORMAL FORM (CNF)
- GREIBACH NORMAL FORM (GNF)

Definition of Chomsky Normal Form:
A CFG is in Chomsky Normal Form if the Productions are in the following forms:

- $A \rightarrow a$
- $A \rightarrow BC$
- $S \rightarrow \epsilon$

where A, B, and C are non-terminals and **a** is terminal.

Algorithm to Convert into Chomsky Normal Form:
Step1 If the start symbol S occurs on some right side, create a new start symbol S' and a new production S' → S.
Step2 Remove Null productions. (Using the Null production removal algorithm discussed earlier)
Step3 Remove unit productions. (Using the Unit production removal algorithm discussed earlier)
Step4 Replace each production $A \rightarrow B_1 \ldots Bn$ where $n > 2$ with $A \rightarrow B_1C$ where $C \rightarrow B_2 \ldots Bn$.
Repeat this step for all productions having two or more symbols in the right side.
Step5 If the right side of any production is in the form A→ aB where a is a terminal and A, B are
non-terminal, then the production is replaced by A→ XB and X→a.
Repeat this step for every production which is in the form A→aB.

EXAMPLE:
Convert the following CFG into CNF
S→ ASA|aB, A → B|S, B → b |ϵ

SOLUTIONS:

1. Since **S** appears in R.H.S, we add a new state $\mathbf{S_0}$ and $\mathbf{S_0 \rightarrow S}$ is added to the production set and it becomes:
   $S_0 \rightarrow S$, S→ ASA|aB, A → B|S, B → b |ϵ

2. Now we will remove the null productions: $B \rightarrow \epsilon$ and $A \rightarrow \epsilon$

After removing B → ϵ, the production set becomes:
S_0→S, S→ ASA | aB | a, A → B | S | ϵ, B → b After removing A → ϵ, the production set becomes:
S_0→S, S→ ASA | aB | a | AS | SA | S, A → B | S, B → b

3. Now we will remove the unit productions. After removing S→ S, the production set becomes:

S_0→S, S→ ASA | aB | a | AS | SA, A → B | S, B → b After removing S_0→ S, the production set becomes:
S_0→ ASA | aB | a | AS| SA, S→ ASA | aB | a | AS | SA A → B|S, B →b
After removing A→ B, the production set becomes:
S_0 → ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA A → S | b
B → b
After removing A→ S, the production set becomes:
S_0 → ASA | aB | a | AS| SA, S→ ASA | aB | a | AS | SA A → b |ASA | aB | a | AS | SA, B →b

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

Here, S_0→ ASA, S → ASA, A→ ASA violates two Non-terminals in R.H.S.
Hence we will apply step 4 and step 5 to get the following final production set which is in CNF:
S_0→ AX | aB | a | AS | SA S→ AX | aB | a | AS | SA
A → b |AX | aB | a | AS | SA B → b
X→ SA

5. We have to change the productions S_0→ aB, S→ aB, A→ aB And the final production set becomes:

S_0→ AX | YB | a | AS | SA S→ AX | YB | a | AS | SA
A → b |AX | YB | a | AS | SA B → b
X→ SA
Y → a

Definition of Greibach Normal Form:

A CFG is in Greibach Normal Form if the Productions are in the following forms: A → b
A → bD_1…Dn S → ϵ
where A, D_1,....,D_n are non-terminals and **b** is a terminal.

Algorithm to Convert a CFG into Greibach Normal Form:
Step1 If the start symbol **S** occurs on some right side, create a new start symbol
S' and a new production **S'** → **S**.
Step2 Remove Null productions. (Using the Null production removal algorithm discussed earlier)

Step3 Remove unit productions. (Using the Unit production removal algorithm discussed earlier)
Step4 Remove all direct and indirect left-recursion.
Step5 Do proper substitutions of productions to convert it into the proper form of GNF.

PROBLEM:

Convert the following CFG into CNF
S→ XY | Xn | p
X → mX | m
Y → Xn | o

SOLUTION:

Here, **S** does not appear on the right side of any production and there are no unit or null productions in the production rule set. So, we can skip Step 1 to Step 3.
Step 4**:**
Now after replacing X in S → XY | Xo | p
With mX | m
we obtain
S → mXY | mY | mXo | mo | p.
And after replacing
X in Y→ Xn | o with the right side of
X → mX | m we obtain
Y→ mXn | mn | o.
Two new productions O→ o and P → p are added to the production set and then we came to the final GNF as the following:
S → mXY | mY | mXC | mC | p X→ mX | m
Y→ mXD | mD | o O → o
P → p

LECTURE 4: PROPERTIES OF CONTEXT FREE GRAMMAR

PUMPING LEMMA FOR CFG:

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 PROPERTY:

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

- Union
- Concatenation
- Kleen Star operation

Union
Let L_1 and L_2 be two context free languages. Then $L_1 \cup L_2$ is also context free.
Example:
Let L_1 = { a^nb^n , n>0}. Corresponding grammar G_1 will have P: S1 → aAb|ab Let L_2 = { c^md^m , n≥0}. Corresponding grammar G_2 will have P: S2→ cBb| ε Union of L_1 and L_2, L = $L_1 \cup L_2$ = { a^nb^n } ∪ { c^md^m }
The corresponding grammar G will have the additional production S → S1 | S2

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

Example:

Union of the languages L_1 and L_2, $L = L_1L_2 = \{ a^nb^nc^md^m \}$
The corresponding grammar G will have the additional production S → S1 S2

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

Example:

Let $L = \{ a^nb^n , n\geq 0\}$. Corresponding grammar G will have P: S → aAb| ε Kleen Star $L_1 = \{ a^nb^n \}^*$
The corresponding grammar G_1 will have additional productions $S_1 \rightarrow SS_1 \mid \varepsilon$
Context-free languages are **not closed** under:
Intersection : If L1 and L2 are context free languages, then L1∩ L2 is not necessarily context free.

Intersection with Regular Language: If L1 is a regular language and L2 is a context free language, then L1 ∩L2 is a context free language.
Complement : If L1 is a context free language, then L1' may not be context free.

LECTURE 5: PUSH DOWN AUTOMATA

PUSH DOWN AUTOMATA:

INTRODUCTION:

As we have seen, Finite automata cannot recognize context free languages (e.g., $\{a^n b^n \mid n \geq 1\}$). It happens because of having finite memories. Whereas the recognition of a context free language may require storing of an unbounded amount of information. For example when we scanning a string from above mentioned language then our checking procedure will be of two folds:(i) All the a's must precede the occurrence of first b. (ii) total counting of the number of a's must be equal with the number of b's. Since n is unbounded, this counting cannot be done with a finite memory.

Hence we need a machine that can count without limit, and thus leading to a new class of automata: Push Down Automata (PDA).

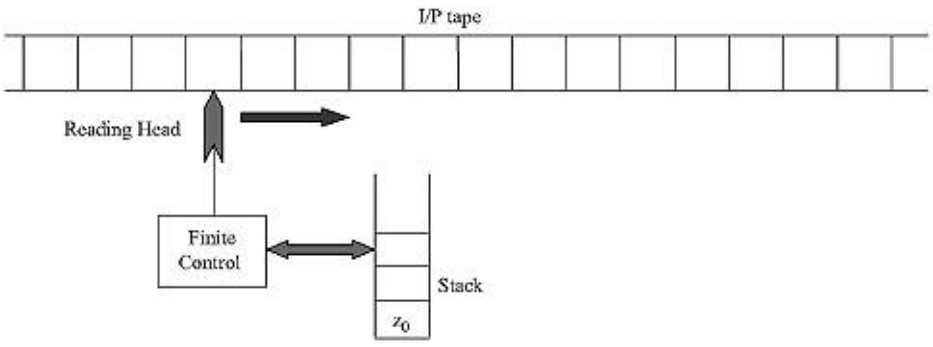


Figure 3.1: Block diagram of Push down Automata

Before giving a formal definition, let us consider component of Push Down Automata and the way it operates depicted in figure 3.1 . Any pda has a read only input tape, an input alphabet, a finite state control, a set of final states, and an initial state as in the case of a finite automata. In addition to these it has a stack called the pushdown store (PDS). It is a read-write push down store as we add elements to PDS or remove elements from PDS. A finite automaton being present in some state and on reading an input symbol either moves to a new state or remains in the same state. Similarly the pda being present in a state and after reading an input symbol and the topmost symbol in pds it moves either to a new state or remains on the same state and writes (add) some symbols in push down store.

Formally a pda can be represented by a 7-tuple as follows: $(\mathbf{Q, \sum, \Gamma, \delta, q_0, Z_0, F})$

Where,

i. Q is a finite non-empty set of states,
ii. $\sum$is a finite non-empty set of symbols,
iii. Γ is a finite non-empty set of push down symbols,
iv. $q_0$is the initial state,
v. F is a set of final set which is a subset of Q,
vi. Z_0 is a special push down symbol called the initial symbol on the push down store.
vii. $\delta$is the transition function from $Q \times (\sum \cup \{\Lambda\}) \times \Gamma$ **to the finite subset of** $Q \times \Gamma^*$.

The steps to write moves of PDA:

a) To PUSH a symbol onto the STACK:
$\delta(q_0, a, Z_0) = \{q_0, aZ_0\}$
The above line is required to store 'a' in empty pds.

b) To POP a symbol from the STACK:
$\delta(q_0, b, a) = \{ q_1, \mathbf{\Lambda}\}$

when we received an input 'b' in state $q_0$and stack top element contain 'a' then our pda will change from state q_0 to q_1 using the above rule.

c) Λ-Moves:
$\delta(q_0, \mathbf{\Lambda}, Z_0) = \{q_0, \mathbf{\Lambda}\}$

Instantaneous Description:
In case of finite automata, it is enough to specify the current state at any time and the remaining input string to be processed. But in case of pda as we have one additional structure namely pds, we have to specify the current state, the remaining input string to be processed and the symbol in the pds.
Thus introducing the next definition:
Let A= $(\mathbf{Q,\sum,\Gamma,\delta, q_0, Z_0, F})$ be a pda. An ID is $(\mathbf{q, x, \alpha})$ where $q \in Q$, $x \in \sum^*$ **and** $\alpha \in \mathbf{\Gamma^*}$.

Ex: ($\mathbf{q_1}$, abaab , bb$\mathbf{Z_0}$ **)** is an ID.

Initial ID:
(q_0, x, Z_0) is called the initial ID. Here x is any input string to be processed and x can be empty **(Λ). If it is Λ then it is called** Λ-Moves.

Ex 1. Design a PDA for the language $L=\{a^n b^n \mid n \geq 1\}$accepted by push down store.

The pda is defined as follows:
$A=(\{q_0,q_1\}, \{a,b\}, \{a, Z_0\},\delta,q_0,Z_0, F)$

where $F=\phi$ and

δ is defined as follows:

R1: $\delta(q_0, a, Z_0) = \{q_0, aZ_0\}$ // R1 is required to store 'a' in empty pds.
R2: $\delta(q_0, a, a) = \{q_0, aa\}$ // R2 is required to store 'a^n' in pds.
R3: $\delta(q_0, b, a) = \{q_1, \mathbf{\Lambda}\}$ // when we encounter first 'b' in q_0 then our pda will change from state q_0 to q_1 using R3.
R4: $\delta(q_1, b, a) = \{q_1, \mathbf{\Lambda}\}$ // In state q_1 if we encounter b's then we need to pop 'a' using rule R4.

R5: $\delta(q_1, \mathbf{\Lambda}, Z_0) = \{q_1, \mathbf{\Lambda}\}$ // After processing whole input string to erase Z0 we need rule R5.

Let consider one input string w=aaabbb ,and check whether the rules are working properly or not.

PROCESSED INPUT	UNREAD INPUT	STACK CONTENT	STATE	RULE USED
--	aaabbb	Z_0	q_0	--
a	aabbb	aZ_0	q_0	R1
a	abbb	aaZ_0	q_0	R2
a	bbb	$aaaZ_0$	q_0	R2
b	bb	aaZ_0	q_1	R3
b	b	aZ_0	q_1	R4
b	**Λ**	Z_0	q_1	R4
Λ	**Λ**	**Λ**	q_1	R5

Conversion of PDA from given CFG :

Theorem: If L is a context-free language, then we can construct a pda A accepting L by empty store, i.e. L=N(A).

Construction of A is defined as follows:

Ex1: Construct a PDA 'A' equivalent to the following CFG: {S→OBB, B→0S/1S/0}

Define pda A as follows:

$A = (\{q\}, \{0,1\}, \{S,B,0,1\}, \delta, q, S, \phi)$

Where, δ is defined as the following rules:

R1: $\delta(q, \Lambda, S) = \{q, 0BB\}$

R2: $\delta(q, \Lambda, B) = \{(q, 0S), (q,1S), (q,0)\}$

R3: $\delta(q, 0, 0) = \{(q, \Lambda)\}$

R4: $\delta(q, 1, 1) = \{(q, \Lambda)\}$

MODULE 5: TURING MACHINE

LECTURE 1: INTRODUCTION TO TURING MACHINE

INTRODUCTION:

A Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars. It was invented in 1936 by Alan Turing.

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape dividedintocellsonwhichinputisgiven.Itconsistsofaheadwhichreadstheinputtape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

FORMAL DEFINITION:

A TM can be formally described as a 7-tuple (Q, X, Σ, δ, q0, B, F) where:

- **Q** is a finite set of states
- **X** is the tape alphabet
- **Σ** is the input alphabet
- **δ** is a transition function; $\delta : Q \times X \rightarrow Q \times X \times \{Left\_shift, Right\_shift\}$.
- $\mathbf{q_0}$is the initial state
- **B** is the blank symbol
- **F** is the set of final states

ACCEPTED LANGUAGE AND DECIDED LANGUAGE:

A TM accepts a language if it enters into a final state for any input string **w**. A language is recursively enumerable (generated by Type-0 grammar) if it is accepted by a Turing machine.

A TM decides a language if it accepts it and enters into a rejecting state for any input not in the language. A language is recursive if it is decided by a Turing machine.

There may be some cases where a TM does not stop. Such TM accepts the language, but it does not decide it.

LECTURE 2: DESIGN PROCEDURE OF A TURING MACHINE

DESIGN PROCEDURE OF A TURING MACHINE

The basic guidelines of designing a Turing machine have been explained below with the help of a couple of examples.

EXAMPLE I:

Design a TM to recognize all strings consisting of an odd number of **α**'s.

SOLUTION:

The Turing machine **M** can be constructed by the following moves:

Let $\mathbf{q_1}$be the initial state.
If **M** is in $\mathbf{q_1}$; on scanning **α**, it enters the state $\mathbf{q_2}$and writes **B**(blank).

If **M** is in $\mathbf{q_2}$; on scanning **α**, it enters the state $\mathbf{q_1}$and writes **B**(blank).

From the above moves, we can see that **M** enters the state $\mathbf{q_1}$if it scans an even number of **α**'s, and it enters the state $\mathbf{q_2}$ if it scans an odd number of **α**'s. Hence $\mathbf{q_2}$ is the only accepting state.

Hence,

$M = \{\{q_1, q_2\}, \{1\}, \{1, B\}, \delta, q_1, B, \{q_2\}\}$

where **δ** is given by:

Tape alphabet symbol	Present State 'q1'	Present State 'q2'
α	BRq_2	BRq_1

TABLE 4.1: TRANSITION TABLE

EXAMPLE II:

Design a Turing Machine that reads a string representing a binary number and erases all leading 0's in the string. However, if the string comprises of only 0's, it keeps one 0.

SOLUTION:

Let us assume that the input string is terminated by a blank symbol, B, at each end of the string.

The Turing Machine, M, can be constructed by the following moves:

Let $\mathbf{q_0}$be the initial state.

If **M** is in $\mathbf{q_0}$, on reading 0, it moves right, enters the state $\mathbf{q_1}$and erases 0. On reading 1, it enters the state $\mathbf{q_2}$and moves right.

If **M** is in $\mathbf{q_1}$, on reading 0, it moves right and erases 0, i.e., it replaces 0's by B's. On reaching the leftmost 1, it enters $\mathbf{q_2}$and moves right. If it reaches B, i.e., the string comprises of only 0's, it moves left and enters the state$\mathbf{q_3}$.

If **M** is in $\mathbf{q_2}$, on reading either 0 or 1, it moves right. On reaching B, it moves left andentersthestate$\mathbf{q_4}$.Thisvalidatesthatthestringcomprisesonlyof0'sand1's.

If **M** is in $\mathbf{q_3}$, it replaces B by 0, moves left and reaches the final state $\mathbf{q_f}$.

If **M** is in $\mathbf{q_4}$, on reading either 0 or 1, it moves left. On reaching the beginning of the string, i.e., when it reads B, it reaches the final state $\mathbf{q_f}$.

Hence,

$$M = \{\{q_0, q_1, q_2, q_3, q_4, q_f\}, \{0,1, B\}, \{1, B\}, \delta, q_0, B, \{q_f\}\}$$

where **δ** is given by(in table 4.2):

Tape alphabet symbol	**Present State 'q0'**	**Present State 'q1'**	**Present State 'q2'**	**Present State 'q3'**	**Present State 'q4'**
0	BRq_1	BRq_1	$0Rq_2$	-	$0Lq_4$
1	$1Rq_2$	$1Rq_2$	$1Rq_2$	-	$1Lq_4$
B	BRq_1	BLq_3	BLq_4	$0Lq_f$	BRq_f

TABLE 4.2: TRANSITION TABLE

LECTURE 3: CHURCH TURING HYPOTHESIS

CHURCH TURING HYPOTHESIS:

The Church-Turing thesis (formerly commonly known simply as Church's thesis) says that any real-world computation can be translated into an equivalent computation involving a Turing machine. In Church's original formulation (Church 1935, 1936), the thesis says that real-world calculation can be done using the lambda calculus, which is equivalent to using general recursive functions.

The Church-Turing thesis encompasses more kinds of computations than those originally envisioned, such as those involving cellular automata, combinators, register machines, and substitution systems. It also applies to other kinds of computations found in theoretical computer science such as quantum computing and probabilistic computing.

There are conflicting points of view about the Church-Turing thesis. One says that it can be proven, and the other says that it serves as a definition for computation. There has never been a proof, but the evidence for its validity comes from the fact that every realistic model of computation, yet discovered, has been shown to be equivalent. If there were a device which could answer questions beyond those that a Turing machine can answer, then it would be called an oracle.

Some computational models are more efficient, in terms of computation time and memory, for different tasks. For example, it is suspected that quantum computers can perform many common tasks with lower time complexity, compared to modern computers, in the sense that for large enough versions of these problems, a quantum computer would solve the problem faster than an ordinary computer. In contrast, there exist questions, such as the halting problem, which an ordinary computer cannot answer, and according to the Church-Turing thesis, no other computational device can answer such a question.

The Church-Turing thesis has been extended to a proposition about the processes in the natural world by Stephen Wolfram in his principle of computational equivalence (Wolfram 2002), which also claims that there are only a small number of intermediate levels of computing power before a system is universal and that most natural systems are universal.

LECTURE 4: TYPES OF TURING MACHINE

TYPES OF TURING MACHINE

LINEAR BOUNDED AUTOMATA:

Alinearboundedautomatonisamulti-tracknon-deterministicTuringmachinewithatape of some bounded finite length.
Length = function (Length of the initial input string, constant c)
Here,
Memory information ≤ c × Input information
The computation is restricted to the constant bounded area. The input alphabet contains two special symbols which serve as left end markers and right end markers which mean the transitions neither move to the left of the left end marker nor to the right of the right end marker of the tape.

FORMAL DEFINITION:

A linear bounded automaton can be defined as an 8-tuple (Q, X, Σ, q0, M_L, M_R, δ, F) where:

- **Q** is a finite set of states
- **X** is the tape alphabet
- **Σ** is the input alphabet
- **q_0**is the initial state
- **M_L** is the left end marker
- **M_R** is the right end marker where $M_R \neq M_L$

δ is a transition function which maps each pair (state, tape symbol) to (state, tape symbol, Constant '**c**') where **c** can be 0 or +1 or-1

F is the set of final states

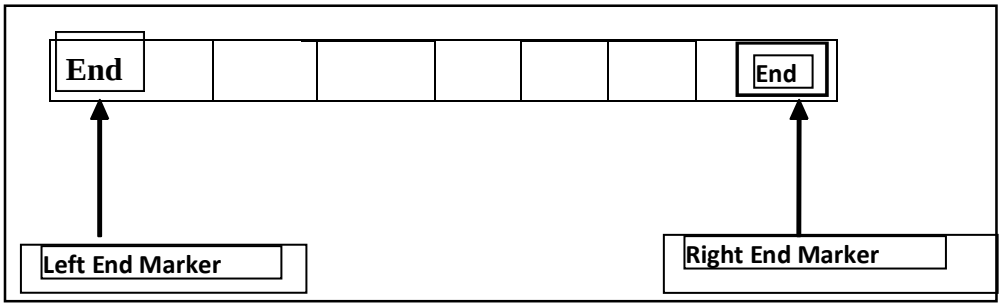


FIG 4.1: LINEAR BOUNDED AUTOMATA

A deterministic linear bounded automaton is always **context-sensitive** and the linear bounded automaton with empty language is **undecidable**.

MULTI TAPE TURING MACHINE

Multi-tape Turing Machines have multiple tapes where each tape is accessed with a separate head. Each head can move independently of the other heads. Initially the input is on tape 1 and others are blank. At first, the first tape is occupied by the input and the other tapes are kept blank. Next, the machine reads consecutive symbols under its heads and the TM prints a symbol on each tape and moves its heads.

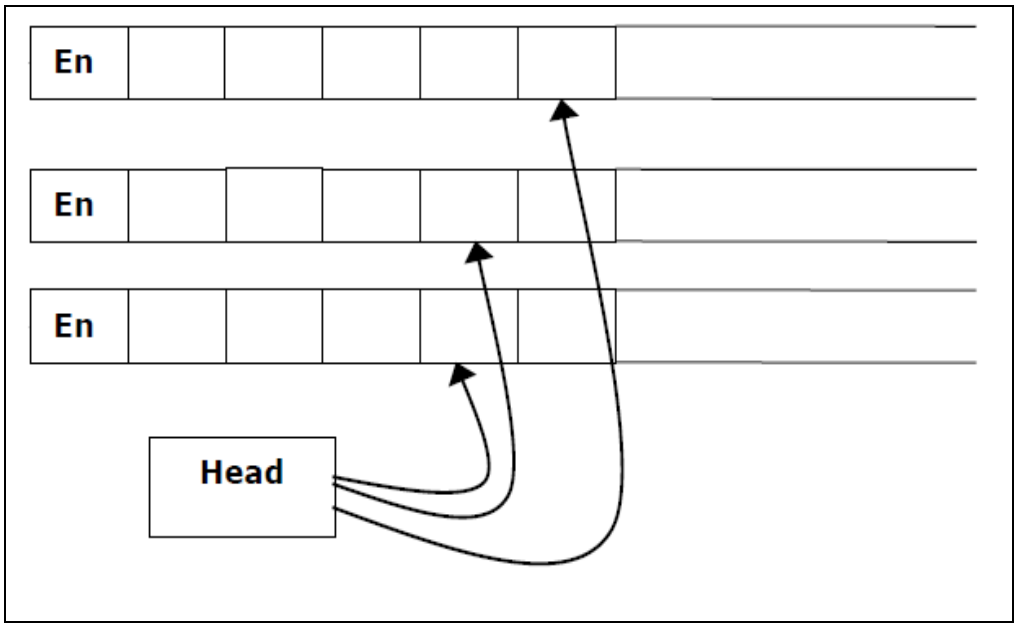


FIG 4.2: MULTI TAPE TURING MACHINE

and the TM prints a symbol on each tape and moves its heads.

A Multi-tape Turing machine can be formally described as a 6-tuple (Q, X, B, δ, q0, F) where:

- **Q** is a finite set of states
- **X** is the tape alphabet
- **B** is the blank symbol
- **δ** is a relation on states and symbols where
  δ: $Q \times X^k \rightarrow Q \times (X \times \{Left\_shift, Right\_shift, No\_shift\})^k$ where there is **k** number of tapes
- $\mathbf{q_0}$ is the initial state
- **F** is the set of final states

LECTURE 5: TURING MACHINE HALTING PROBLEM

TURING MACHINE HALTING PROBLEM:

Input: A Turing machine and an input string **w**.

Problem: Does the Turing machine finish computing of the string **w** in a finite number of steps? The answer must be either yes or no.

Proof: At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a **Halting machine** that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as 'yes', otherwise as 'no'. The following is the block diagram of a Halting machine:

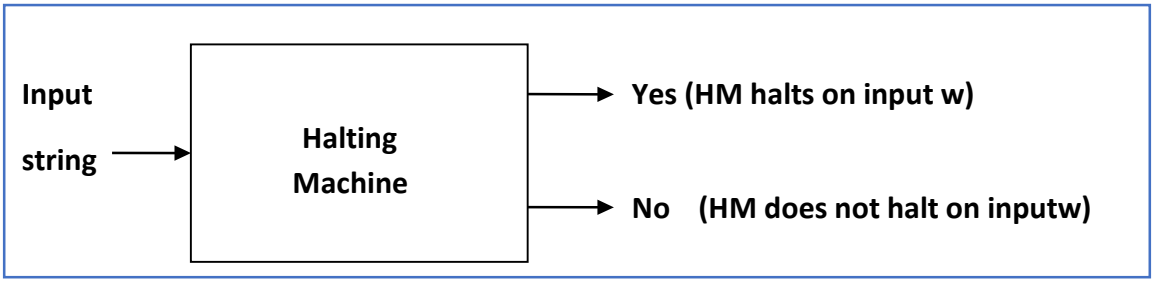


FIG 4.3: BLOCK DIAGRAM OF HALTING MACHINE

Now we will design an **inverted halting machine (HM)'** as:

If **H** returns YES, then loop forever.
If **H** returns NO, then halt.

The following is the block diagram of an 'Inverted halting machine':

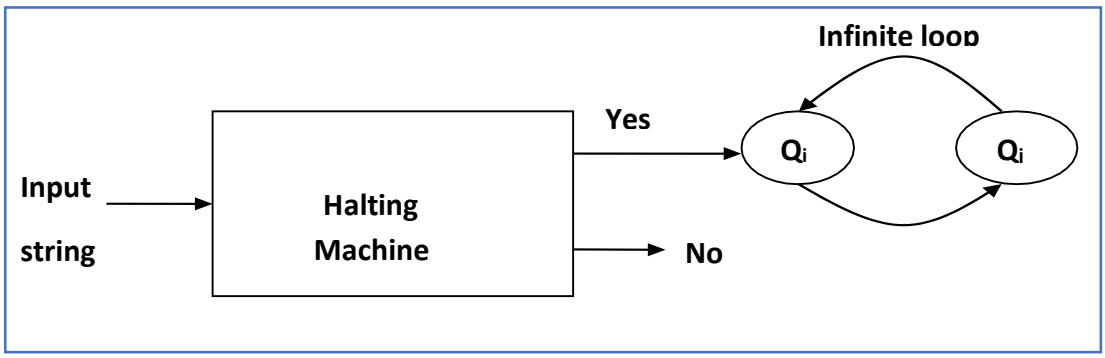


FIG 4.4: INVERTED HALTING MACHINE

Further, a machine **$(HM)_2$** which input itself is constructed as follows:

- If $(HM)_2$ halts on input, loop for ever.
- Else, halt.

Here, we have got a contradiction. Hence, the halting problem is **undecidable.**