NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC A Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIALS(2019 SCHEME)

CST 301 FORMAL LANGUAGES AND AUTOMATA THEORY

VISION OF THE INSTITUTION

To Mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the University of Dr. A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES

SUBJECT CODE: C301		
COURSE OUTCOMES		
C301.1	K6	**Identify** and **construct** the various automata for Regular languages.
C301.2	K5	**Explain** the different properties and representations of Regular languages
C301.3	K5	**Explain** the different representation of Regular Language and Context Free Languages
C301.4	K6	**Design** pushdown automata for context free languages
C301.5	K6	**Explain** the representation of context sensitive language and **Design** the Turing machine for recursively enumerable language

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

CO'S	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C301.1	3	3	3	-	-	-	-	-	-	-	-	3
C301.2	3	3	3	-	-	-	-	-	-	-	-	3
C301.3	3	3	3	-	-	-	-	-	-	-	-	3
C301.4	3	3	3	3	-	-	-	-	-	-	-	3
C301.5	3	3	3	3	-	-	-	-	-	-	-	3
C301	3	3	3	3	-	-	-	-	-	-	-	3

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

PSO MAPPINGS

CO'S	PSO1	PSO2	PSO3
C301.1	3	-	-
C301.2	3	2	-
C301.3	3	2	-
C301.4	3	2	-
C301.5	3	2	-
C301	3	2	-

SYLLABUS

Syllabus

CST 301 Formal Languages and Automata Theory

Module - 1 (Introduction to Formal Language Theory and Regular Languages)

Introduction to formal language theory– Alphabets, Strings, Concatenation of strings, Languages.

Regular Languages - Deterministic Finite State Automata (DFA) (Proof of correctness of construction not required), Nondeterministic Finite State Automata (NFA), Equivalence of DFA and NFA, Regular Grammar (RG), Equivalence of RGs and DFA.

Module - 2 (More on Regular Languages)

Regular Expression (RE), Equivalence of REs and DFA, Homomorphisms, Necessary conditions for regular languages, Closure Properties of Regular Languages, DFA state minimization (No proof required).

Module - 3 (Myhill-Nerode Relations and Context Free Grammars)

Myhill-Nerode Relations (MNR)- MNR for regular languages, Myhill-Nerode Theorem (MNT) (No proof required), Applications of MNT.

Context Free Grammar (CFG)- CFG representation of Context Free Languages (proof of correctness is required), derivation trees and ambiguity, Normal forms for CFGs.

Module - 4 (More on Context-Free Languages)

Nondeterministic Pushdown Automata (PDA), Deterministic Pushdown Automata (DPDA), Equivalence of PDAs and CFGs (Proof not required), Pumping Lemma for Context-Free Languages (Proof not required), Closure Properties of Context Free Languages.

Module - 5 (Context Sensitive Languages, Turing Machines)

Context Sensitive Languages - Context Sensitive Grammar (CSG), Linear Bounded Automata.

Turing Machines - Standard Turing Machine, Robustness of Turing Machine, Universal Turing Machine, Halting Problem, Recursive and Recursively Enumerable Languages.

Chomsky classification of formal languages.

Text Book

1. Dexter C. Kozen, Automata and Computability, Springer (1999)

Reference Materials

1. John E Hopcroft, Rajeev Motwani and Jeffrey D Ullman, Introduction to Automata Theory, Languages, and Computation, 3/e, Pearson Education, 2007
2. Michael Sipser, Introduction To Theory of Computation, Cengage Publishers, 2013.

Sample Course Level Assessment Questions

Course Outcome 1 (CO1): Identify the class of the following languages in Chomsky Hierarchy:

- $L_1 = \{a^p | p \text{ is a prime number}\}$
- $L_2 =$

$\{x\{0,1\}^* | x \text{ is the binary representation of a decimal number which is a multiple of } 5\}$

- $L_3 = \{a^n b^n c^n | n \geq 0\}$
- $L_4 = \{a^m b^n c^{m+n} | m > 0, n \geq 0\}$
- $L_5 = \{M\#x | M \text{ halts on } x\}$. Here, M is a binary encoding of a Turing Machine and x is a binary input to the Turing Machine.

Course Outcome 2 (CO2):

(i) Design a DFA for the language $L = \{axb | x \in \{a,b\}^*\}$

(ii) Write a Regular Expression for the language: $L = \{x \in \{a,b\}^* | \text{third last symbol in } x \text{ is } b\}$

(iii) Write a Regular Grammar for the language: $L = \{x \in \{0,1\}^* | \text{there are no consecutive zeros in } x\}$

(iv) Show the equivalence classes of the canonical Myhill-Nerode relation induced by the language: $L = \{x \in \{a,b\}^* | x \text{ contains even number of } a\text{'s and odd number of } b\text{'s}\}$.

Course Outcome 3 (CO3):

(i) Design a PDA for the language $L = \{ww^R | w \in \{a,b\}^*\}$. Here, the notation w^R represents the reverse of the string w.

(ii) Write a Context-Free Grammar for the language $L = \{a^n b^{2n} | n \geq 0\}$.

Course Outcome 4 (CO4):

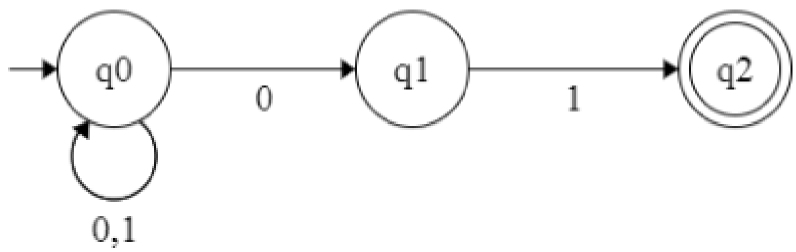
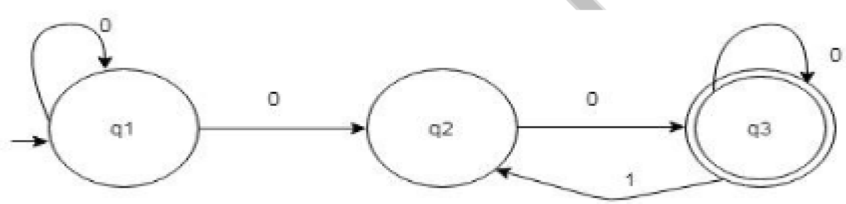
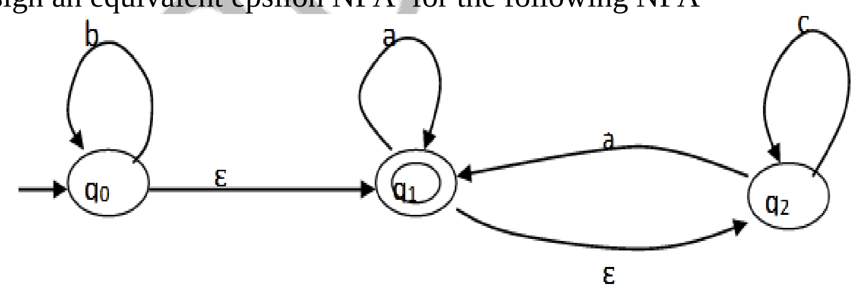
(i) Design a Turing Machine for the language $L = \{a^n b^n c^n | n \geq 0\}$

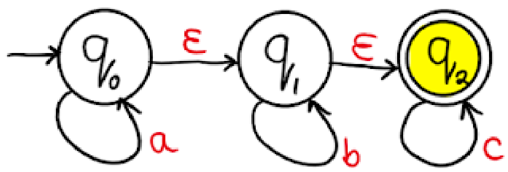
(ii) Design a Turing Machine to compute the square of a natural number. Assume that the input is provided in unary representation.

Course Outcome 5 (CO5): Argue that it is undecidable to check whether a Turing Machine M enters a given state during the computation of a given input x.

QUESTION BANK

MODULE I

Sl.No	Questions	KL/COL
1	Formally define extended delta for an NFA. Show the processing of input w **= 0101** for the following NFA.  	K1/CO1
2	Compare the transition function of DFA,NFA and epsilon-NFA	K5/CO1
3	Convert the following NFA to DFA.  	K3/CO1
4	Design an equivalent epsilon NFA for the following NFA  	K6/CO1
5	Design a DFA for the language {w ε {a, b}* | w is a set of all strings ends with ab}.	K6/CO1
6	Design non deterministic automata (without ε moves) for the regular language that consist of all strings with at least two consecutive 0's. Assume Σ = {0, 1}.	K6/CO1
7	a) Find the ε-closure of each state. (5 mark)	K5/CO1

<table>
<tr><td></td><td>

b) Prove the equivalence of NFA and DFA (5 mark)</td><td></td></tr>
<tr><td>8</td><td>Convert the following NFA to DFA and describe the language it accepts.
$M = (\{P, Q, R, S, T\}, \{0,1\}, \delta, P, \{S, T\})$ and δ is given as:
<table>
<tr><th></th><th>0</th><th>1</th></tr>
<tr><td>P</td><td>{P,Q}</td><td>{P}</td></tr>
<tr><td>Q</td><td>{R,S}</td><td>{T}</td></tr>
<tr><td>R</td><td>{P,R}</td><td>{T}</td></tr>
<tr><td>S</td><td>-</td><td>-</td></tr>
<tr><td>T</td><td>-</td><td>-</td></tr>
</table></td><td>K3/CO1</td></tr>
<tr><td>9</td><td>Construct a DFA over {a,b} which accepts all strings which has even number of a’s and even number of b’s</td><td>K6/CO1</td></tr>
</table>

MODULE II

Sl.No	Questions	KL/COL
1	Design regular expression for the language that consists of all strings ending with 00. Assume Σ= {0, 1}.	K6/CO2
2	Define nondeterministic finite automata (NFA). Draw the NFA for the language $L=\{a^n b^m \mid n, m>=1\}$	K1/CO2
3	Construct non deterministic finite automata (with ε moves) for regular expression (0+1)*1.	K6/CO2
4	Give the regular expression for the language: strings of ‘a’ and ‘b’ containing at least two ‘b’.	K1/CO2
5	Find an ε-NFA for the language L (a(a+b)*b) employing the rules for regular expression to ε-NFA conversion.	K3/CO2
6	Prove the equivalence of nondeterministic finite automata with ε moves and regular expressions.	K6/CO2
7	State the closure properties of regular Languages.	K1/CO2

8	Find an equivalent ε-NFA for the following regular expression $(0+1)^*011$	K3/CO2
9	Find an equivalent ε-NFA for the following regular expression 1(0+1) *0	K3/CO2
10	Define nondeterministic finite automata (NFA). Draw the NFA for the language over {0,1}in which the string contain either 2 consecutive 0's or 2 consecutive 1's.	K1/CO2

MODULE III

Sl.No	Questions	KL/COL
1	Define context free grammar. Derive any two representative strings with minimum length 4 from following context free grammar. G= ({S,A,B},{a,b},P,S) **S→bA | aB** **A→bAA | aS | a** **B→aBB | bS | b**	K2/CO3
2	Construct the CFG for the language having any number of a's over the set ∑= {a}.	K6/CO3
3	Construct a CFG for the regular expression (0+1)*	K6/CO3
4	Construct a CFG for a language L = (a+b)*| where w € (a, b)*}.	K6/CO3
5	Construct the CFG for the language having any number of 0's over the set ∑= {0}.	K6/CO3
6	Convert the following CFG to CNF S-> ABA | BaA | A A-> Ba | S| € B-> Ba |b | Ca C-> Ca D-> DaD |a	K2/CO3
7	What do you mean by useless symbol in a grammer. List the condition for symbols to become useful symbols in CFG	K1/CO3
8	Eliminate Useless symbol for the following Production S->AB|a A->BC|b B->aB|C C-> aC|B	K4/CO3
9	Convert the following CFG to GNF **S-> CA | BB** **B-> b | SB**	K2/CO3

	C-> b **A-> a**	
10	Eliminate € production for the following Production **S->AB** **A->aAA | €** **B->bBB | €**	K4/CO3
11	Show that the grammar S->SS|a|b is ambiguous for the string aaba	K3/CO3

MODULE IV

Sl.No	Questions	KL/COL
1	Write the closure properties of CFL	K1/CO5
2	Show that $a^n b^n c^n$ |n>=0 is not a CFL	K3/CO5
3	Construct the PDA for the given CFG and test whether 010000 is acceptable by PDA S->0BB B->0S|1S|0	K6/CO5
4	Design PDA to accept the palindrome of the form WCW^R over $(a,b)^+$ along with instantaneous descriptor of the same	K6/CO5
5	Construct a PDA to accept the language $a^n b^m c^n$ | n,m>=1	K6/CO5
6	Show that L={a^p|p ia a prime} is not a CFL	K3/CO5
7	Construct the PDA to accept the language $a^{m+n} b^m c^n$	K6/CO5
8	Construct the PDA for the given CFG and test whether aabaaa is acceptable by PDA **S->aAA** **A->aS|bS|a**	K6/CO5
9	Design PDA to accept the string w such that $n_a(w)=n_b(w)$ by **final state** and **empty stack** and also write the instantaneous descriptor of the same	K6/CO5
10	Construct a PDA to accept the language $a^{n+m} b^m c^n$ | n,m>=1	K6/CO5
11	Show that $a^n b^n c^n$ |n>=0 is not a CFL	K3/CO5
12	Construct the PDA to accept the language $a^n b^m c^{n+m}$	K6/CO5
13	Construct the PDA for the given CFG and test whether 001011 is acceptable by PDA	K6/CO5

14	Design PDA to accept the string w such that $a^n b^n$ by **final state** and **empty stack** and also write the instantaneous descriptor of the same	K6/CO5

MODULE V

Sl.No	Questions	KL/COL
1	Construct a TM that recognizes the language 01*0	K6/CO5
2	Define LBA along with its Tuples	K1/CO5
3	Construct a TM to accept a Palindrome	K6/CO5
4	Construct a TM to accept strings over {0,1} in which equal number of 0's and 1's present	K6/CO5
5	Construct a TM to accept the strings over {a,b} in which substring aba is present	K6/CO5
6	Define CSG and write the properties of CSL	K1/CO5
7	Construct a TM to accept the language $a^n b^{2n}$	K6/CO5
8	Construct a TM to accept $a^n b^n c^n \mid n>1$	K6/CO5
9	Construct a TM to perform addition of two numbers	K6/CO5
10	Design a Turing Machine for the language $L=a^n b^{2n+1} \mid n>=1$	K6/CO5
11	Construct a TM to accept the strings over {a,b} in which accepts the language aba*b	K6/CO5
12	Construct a TM to perform subtraction of two numbers m and n $f(m,n)=\begin{cases} m-n & \text{if } m>n \\ 0 & \text{If } m<=n \quad \text{choose } m=3 \text{ and } n=2 \end{cases}$	K6/CO5

APPENDIX I

CONTENT BEYOND SYLLABUS

MODULE I NOTES

Theory of computation.

Theory of computation deals with how efficiently problems can be solved on a model of computation, using an algorithm.

* Theory of computation (TOC) can be divided into 3 main areas.

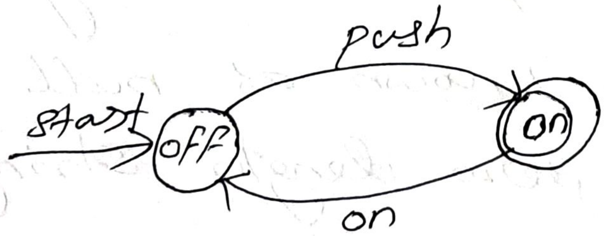
1. Automata theory.
2. computability theory.
3. complexity theory.

Module I

* Introduction to Automata theory.

* Automata are useful model for many important kinds of hardware and software.

Eg: Automaton for an electric switch



central concepts of Automata theory.

1. Alphabet : A finite non-empty set of symbols.

Represented using "Σ".

Egs: $\Sigma = \{0, 1\}$ binary alphabet

$\Sigma = \{a \ldots z\}$ lower case letters.

$\Sigma = \{a, b\}$

$\Sigma = \{a, b, c\}$..

String:

Sequence of symbols chosen from some alphabet.

Egs: 0110
1111
01010 } chosen from binary alphabet

Length of a string "w" is represented using $|w|$. It is actually the number of characters in a string.

Eg: w: 01101

$|w| = 5$

Empty string is represented using "ε"

"ε" is also known as null string or zero length string.

powers of alphabets

Let $\Sigma = \{a, b, c\}$

thus $\Sigma^1 = \{a, b, c\}$ (set of strings of length 1)

$\Sigma^2 = \{a, b, c\} \{a, b, c\}$

$\Rightarrow \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$

(set of strings of length 2)

$\Sigma^+ \Rightarrow \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$

$\Rightarrow \{a, b, aa, ab, ac \dots\}$

$\Sigma^* \Rightarrow \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$

$\Rightarrow \{\varepsilon, a, b, aa, ab, \dots\}$

$+ \Rightarrow$ positive closure

$* \rightarrow$ ~~kleene~~ closure. [Kleen closure]

Language

A set of strings chosen from Σ^*, where Σ is a particular alphabet, is called a language.

eg: L_1 is a language over alphabet $\Sigma = \{0, 1\}$ consists of strings of length 2

$L_1 \Rightarrow \{00, 01, 10, 11\}$

$L_2 :\Rightarrow$ A language over $\Sigma = \{0, 1\}$ consists of equal number of '0's

(4) Followed by equal numbers of 1's

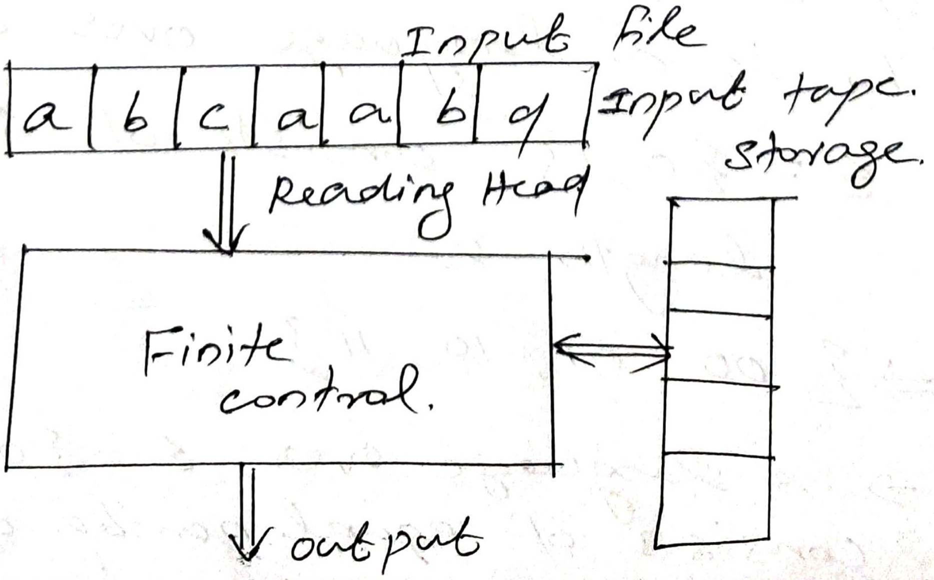
L_2 $\{01, 0011, 000111, \ldots\}$

* FINITE AUTOMATA (FA)

* IG is also known as finite state machines (FSM)
* Finite automata is a mathematical model (Abstract model) used to represent hardware or software.

FA
- without output
  - DFA
  - NFA
  - ε-NFA
- with output
  - Moore machine
  - Mealy Machine

The finite automata can be represented as below



(5) Input tape is a linear tape having some cells which can hold input symbols from Σ

Finite control indicates the current state and decides the next state on receiving a particular input from the tape.

*) when the entire string is read using the read head and if the finite control is in final state then the string is accepted or else it is rejected.

Finite Automata can be represented by a transition diagram in which the vertices represent the states and edges represent the transitions.

Transition Function : Transition is the process of moving from one state to another state on reading an input symbol (denoted by "δ")
Transition function is used to define the rules for moving from one state to another state.

eg: $\delta(x, 1) = y$ this means.

After reading input 1 from the state "x" the automaton moves to "y"

⑥ Transition diagram:

The transition diagram or transition graph is a directed labelled graph in which each vertex or node represents a state and the directed edges show the transition of a state. The Edges are labelled with input.

Notations:

Represents initial state

Represents final state

→ Represents transition from one state to another

* DETERMINISTIC FINITE AUTOMATA (DFA)

* The term Deterministic in DFA refers to the fact that on each input there is one and only one state to which the automaton can transition from its current state.
* DFA has to consume (use) all the input symbols present in Σ.

Definition:

DFA is represented using 5-tuple notation.

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

where A is the name of DFA. we can use any name for a DFA.

$Q \rightarrow$ Finite set of states.

$\Sigma \rightarrow$ Finite set of input symbols. (alphabets)

$\delta \rightarrow$ Transition function. it takes arguments as state and input symbols and returns a state.

$q_0 \rightarrow$ start state, one of the states in Q.

$F \rightarrow$ set of final or Accepting states. The set F is a subset of Q.

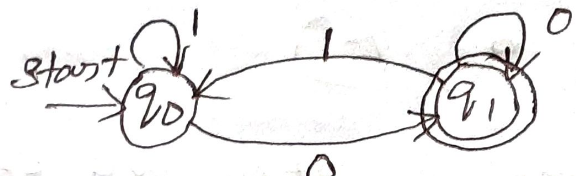
Example:

Design a DFA which accepts a language L over alphabet $\{0,1\}$ such that

$L = \{ w \mid w \text{ ends with } 0 \}$

So here our $L = \{0, 00, 10, 010, 000, 110, \dots\}$

The minimum no. of. states required is 2 since the minimum length string in L is '0' (string length 1).



So this is the DFA which Accepts the strings in L.

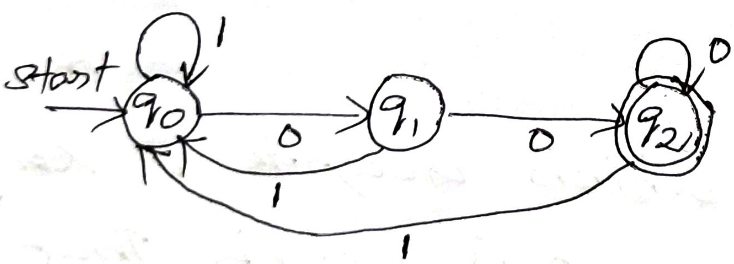
Eg: 2

Design a DFA which accepts strings ending with 00 over $\Sigma = \{0, 1\}$

Here $L = \{$ 00, 000, 100, 0000, 1000, 1100, 0100 ... $\}$

Here the length of smallest string (00) is 2.

So the minimum number of states required in the DFA is $2 + 1 = 3$



Eg: 3

Draw a DFA for the Language accepting strings ending with "01" over alphabet $\Sigma = \{0, 1\}$ $L = \{$ 01, 101, 1101, 0001, 001 ... $\}$

The minimum no. of states $= 2 + 1 = 3$

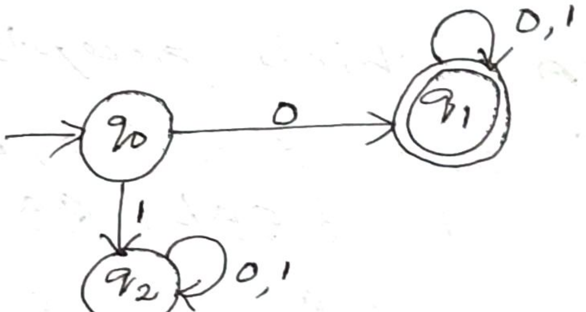
(a)

Eg: 4 (strings start with)

Design a DFA which accepts strings starting with "0" over alphabet $\Sigma = \{0, 1\}$

Here : $L = \{0, 00, 01, 000, 011, 010 \ldots\}$

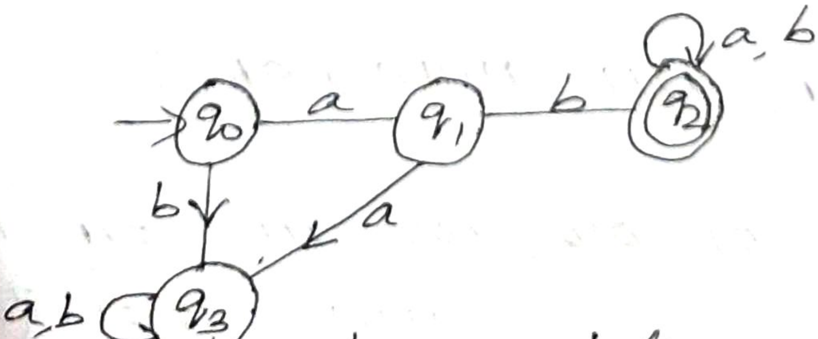
minimum no. of states $= 1 + 1 = 2$



dead state / Trap state.

Eg: 5

Design a DFA which accepts strings starting with ab over alphabet $\Sigma = \{a, b\}$



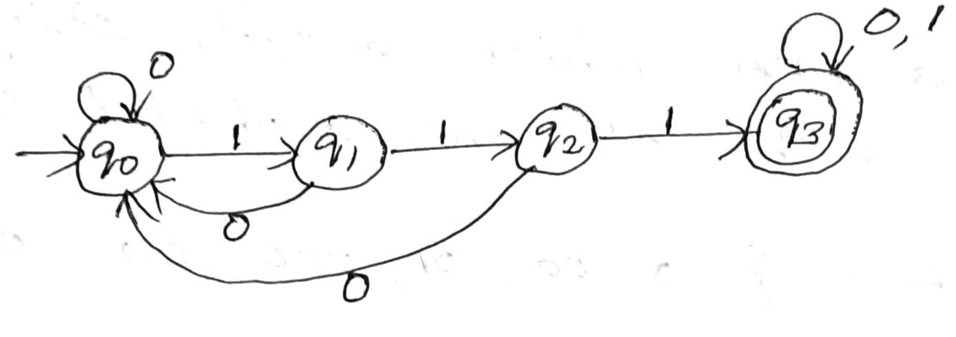
dead state / Trap state

⑩

Eg: 6

* Design a DFA which accepts all the strings contain the substring "111"

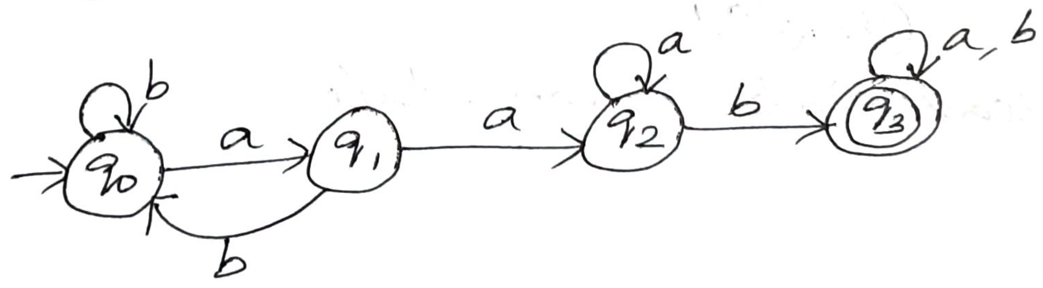
$L = \{ 111, 0111, 1111, 1110, 01110, \ldots \}$



Eg: 7

Design a DFA which accepts a sub-string "aab" over $\Sigma = \{a, b\}$

$L = \{ aab, aabb, baab \ldots \}$



Eg: 8

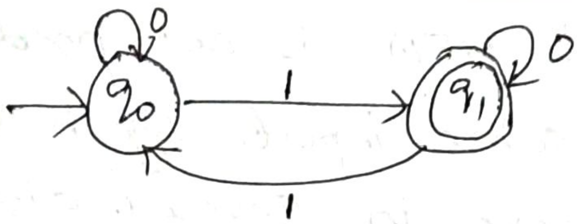
Design a DFA that accepts odd numbers of 1s. $\Sigma = \{0, 1\}$

$L = \{ 1, 001, 11111, 111111, 01, \ldots \}$

$L = \{ 1, 01, 010, 100, 111, 1011, 1101, 11111, 10101, \ldots \}$

odd numbers of 1s.

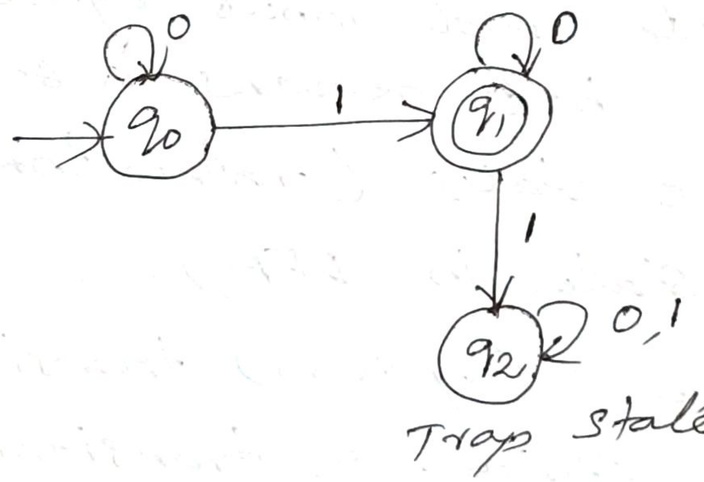
(11)



Eg: 9 (Exactly)

Design a DFA which accepts strings containing exactly one "1" over $\Sigma = \{0, 1\}$

$L = \{1, 01, 10, 100, 001, 010, 1000, \ldots\}$



* Non Deterministic Finite Automata (NFA)

Non-deterministic finite automata (NFA) are similar to DFA except for the fact that rather than a unique transition, NFA allows a set of possible transitions on one single input symbol from the same state.

(12) * The behaviour of NFA is not predictable because on the same input symbol the automaton may move in several directions.

* The designing of an NFA is easier compared to that of DFA and an NFA can be converted to an equivalent DFA.

* NFA is also represented using 5-tuples, $(Q, \Sigma, \delta, q_0, F)$ similar to DFA.

The difference between the DFA and the NFA is in the type of δ.

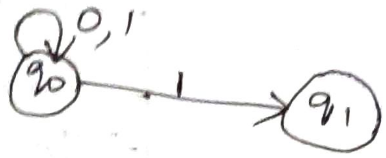
* For the NFA, δ is a function that takes a state and input symbol as arguments but returns a set of 0, 1 or more states (rather than exactly one state in DFA).

* NFA allows ε (epsilon) transitions. ε is empty string. ε transitions is defined as transitions in which NFA can change without reading anything from the i/p string.

In the case of NFA, the δ function is

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

Eg:



Here $\delta(q_0, 1) = \{q_0, q_1\}$

[But in the case of DFA for each δ there will be exactly one state. Here there are 2 states.]

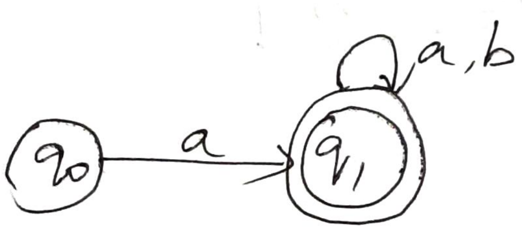
Similarly $\delta(q_1, 0) = \phi$ [The transitions
$\delta(q_1, 1) = \phi$
for 0 and 1 from q_1 is not shown here.]

So it means that from each state for each i/p symbol there are 2^{Q} states possible. [Here it can be

$\phi, q_1, q_2, \{q_1, q_2\}$]

Examples:

1) Design an NFA over an alphabet $\{a, b\}$ such that all the strings start with "a".



$Q = \{q_0, q_1\}$

$\Sigma = \{a, b\}$

$\delta(q_0, a) = q_1$

$\delta(q_1, a) = q_1$

$\delta(q_1, b) = q_1$

$\delta(q_0, b) = \phi$.

initial state q_0.

$F = \{q_1\}$.

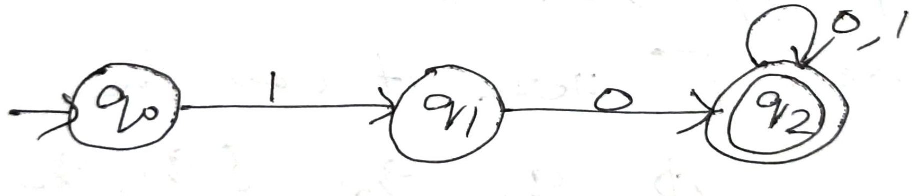
	a	b
$\rightarrow q_0$	q_1	—
$* q_1$	q_1	q_1

→ Here blank can be given or ϕ.

if there is no transition from a state for a particular i/p symbol then that can be left as blank space or ϕ can be used instead (means it is not going anywhere).

Eg 2:

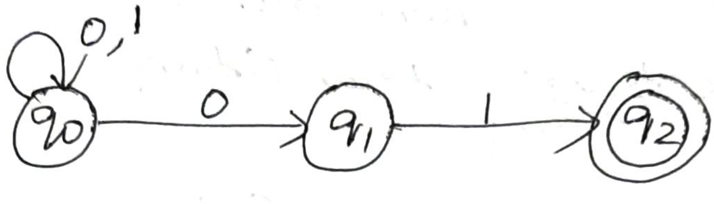
Design an NFA which accepts all the strings starting with 10 over $\{0,1\}$



State/ip	0	1
$\rightarrow q_0$		q_1
q_1	q_2	
$* q_2$	q_2	q_1

Eg 3:

Design an NFA over an alphabet {0,1} such that all the strings ends with "01."



S/ip	0	1
→q_0	q_0, q_1	q_0
q_1		q_2
q_2		

* The difference between DFA and NFA

* DFA is Deterministic Finite Automata	NFA stands for Non deterministic Finite Automata.
* In DFA, for a given state on a given input symbol it goes to exactly one state	* In NFA, there could be multiple states ~~from~~ a particular state for an input symbol.
* For each state, transition on all input symbols (Σ) should be defined.	Not all input symbol transition need to be defined explicitly.

16

DFA	NFA
* DFA does not permit empty string (ϵ) transition.	NFA permits ϵ-transition
* More difficult to design due to deterministic transition.	Easy to design due to non-deterministic transition.
* Here the δ function $\delta : Q \times \Sigma \rightarrow Q$.	Here δ is $\delta : Q \times \Sigma \rightarrow 2^Q$.
* String is accepted by DFA if the transition ends in final state	String is accepted by NFA if one of all possible transition ends in final state.
* All DFAs are NFAs	Not all NFAs are DFAs (So NFA to DFA conversion required).
* DFA requires more space	NFA requires less space than DFA.

NFA to DFA conversion

* An NFA is more of a theoretical concept. So usually an NFA is converted to a DFA for practical implementation.

(17) An NFA can have 0, 1 or more than 1 Transition from a given state on a given input symbol. An NFA can also have Null (ϵ) transitions. DFA has one and only one move from a given state on a given input symbol.

Steps for conversion:

Suppose we have an NFA

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

which recognizes a language L. The DFA $D = (Q', \Sigma, q_0, \delta', F')$ can be constructed for the same language L from the given NFA.

Step 1: initially $Q' = \phi$ (empty set)

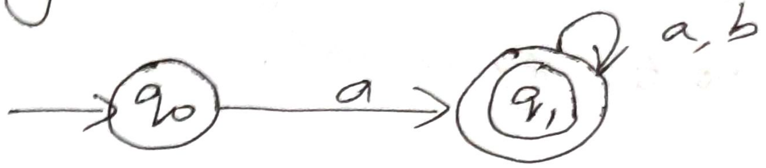
Step 2: Add q_0 (initial state of NFA) to Q'.

Step 3: For each state in Q', find all possible set of states for each input symbol using transition function of NFA (transition table can be used).

If a particular set of states is not in Q' then add it to Q'.

(18) Step 4: Final state of DFA will be states which contains F (final states of NFA)

eg 1: (basic example)



$\Sigma = \{a, b\}$

convert this NFA to DFA.

A. Here $Q = \{q_0, q_1\}$

$\Sigma = \{a, b\}$

Transition function δ is shown in transition table.

state / ip	a	b
$\rightarrow q_0$	q_1	—
$* q_1$	q_1	q_1

Here $q_0 = q_0$ (initial state)

$F = \{q_1\}$.

Now we have to construct DFA from the above NFA.

DFA $(Q', \Sigma, \delta', q_0, F')$.

(19)

* we start from the initial state of NFA (q_0).

So add q_0 to Q'.

Now our $Q' = \{q_0\}$.

Now the transition table is

	a	b
$\to q_0$		

Now take each state from Q'. Here there is only one state in Q' (q_0) and write the transition for that particular state in the above table using transitions from NFA.

	a	b
$\to q_0$	q_1	ϕ

→ Here either you can give ϕ or any state name [whichever you give will be treated as Trap or dead state]

So from the above table we have got 2 more entries (q_1, ϕ(Trap)) which are not in Q'.

So now show the transition for the newly added states. $Q' = \{q_0, q_1, \phi$

	a	b
$\rightarrow q_0$	q_1	ϕ
$*q_1$	q_1	q_1
ϕ	ϕ	ϕ

Trap state.

Here ϕ is the dead / Trap state so that state should be included, you can give any other name to Trap state.

So now check the second and third rows (rows of q_1 and ϕ).

Here we haven't got any new states.

So our $Q' = \{q_0, q_1, \phi\}$

$\Sigma = \{a, b\}$

δ

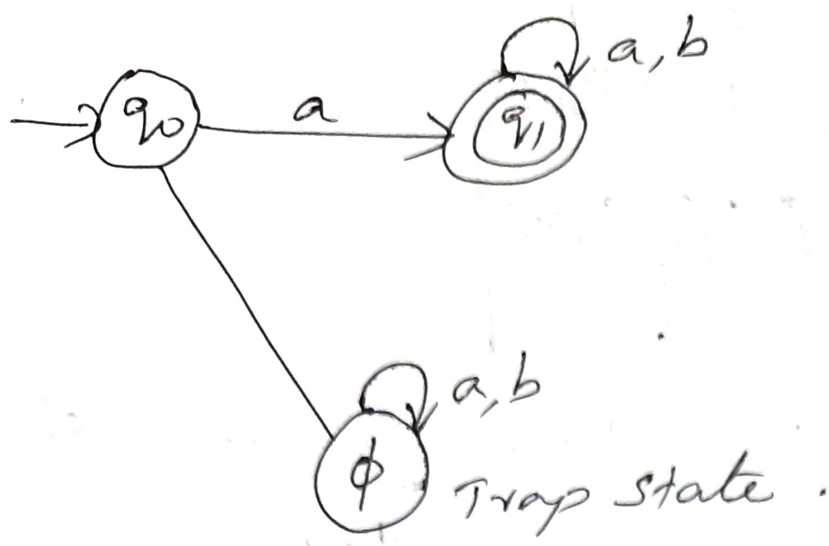
	a	b.
$\rightarrow q_0$	q_1	–
$*q_1$	q_1	q_1
ϕ	ϕ	ϕ ← dead state

$F = \{q_1\}$

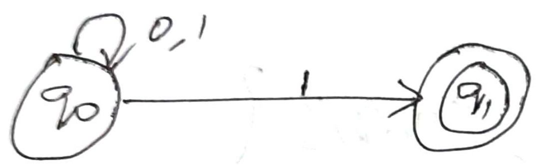
$q_0 = q_0$.

(21)

Now draw the transition diagram.



eg 2:



This is an NFA which accepts a language $L = \{ w / w \text{ ends with } 1 \}$.

Here all the strings w in L ends with 1.

$$L = \{ 1, 01, 11, 111, 011, \ldots \}$$

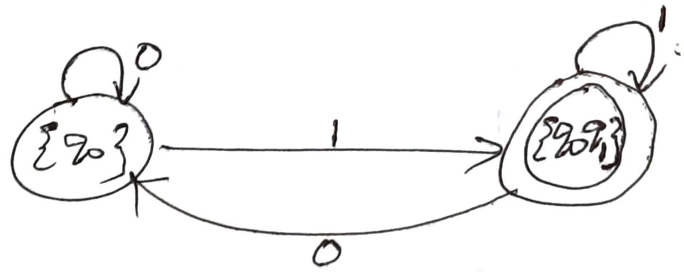
Transition table of NFA

	0	1
q_0	q_0	q_0, q_1
q_1	—	—

DFA

	0	1
$\rightarrow q_0$	$\{q_0\}$	$\{q_0 q_1\}$
$*\{q_0 q_1\}$	$\{q_0\}$	$\{q_0 q_1\}$

Now the transition diagram of DFA



$$Q' = \{\{q_0\}, \{q_0 q_1\}\}$$

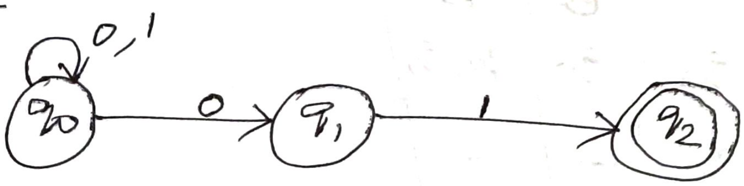
$$F = \{\{q_0 q_1\}\}$$

Eg 3:

convert the below NFA to DFA.

$L = \{$ set of all strings over $\{0,1\}$ that ends with "01" $\}$

NFA



The transition table of NFA is

(23)

	0	1
$\rightarrow q_0$	q_0, q_1	q_0
q_1	—	q_2
$*q_2$	—	—

Now from this we can create the transition table of DFA

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

Here $\delta(q_0 q_1, 0) = \delta(q_0, 0) \cup \delta(q_1, 0)$

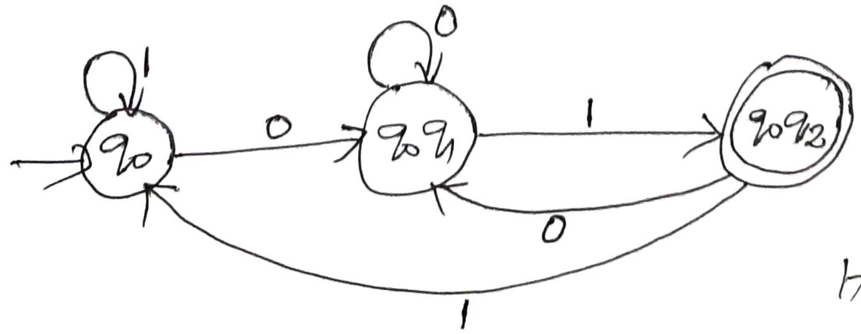
$= \{q_0, q_1\} \cup \phi$

$= \{q_0 q_1\}$

similarly $\delta(q_0 q_2, 0) = \delta(q_0, 0) \cup \delta(q_2, 0)$

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

$= \{q_0 q_1\}$



Now this is a DFA

Here

$Q' = \{q_0, \{q_0 q_1\}, \{q_0 q_2\}\}$

(24) The final state of NFA is q_2.
In DFA we have the state q_0q_2 in which q_2 is present. So that will be the final state of DFA

$$F = \{\{q_0 q_2\}\}$$

* Equivalence of NFA and DFA

Here we have to prove that the DFA obtained from NFA by the conversion method accepts the same language as the NFA.

Theorem:

if $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ is the DFA constructed from NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ by the subset construction, thus

$$L(D) = L(N).$$

ie, $\hat{\delta}_D(q_0, w) = \hat{\delta}_N(q_0, w)$, for an arbitrary string w, we prove this statement. we use induction principle on $|w|$.

Basis of Induction:

Let $|w| = 0$ thus $w = \varepsilon$ (epsilon)

(25) By the definition of extended transition function,

$$\hat{\delta}_D(q_0, \epsilon) = \{q_0\}$$

$$\hat{\delta}_N(q_0, \epsilon) = \{q_0\}$$

ie $\hat{\delta}_D(q_0, \epsilon) = \hat{\delta}_N(q_0, \epsilon)$ is true for $w = \epsilon$.

Inductive step:

Let us assume that the theorem is true for all the strings, w, of length n. ie $|w| = n$. Now let us prove that the same is true for a string of length "$n+1$."

Let $w = xa$ where a is the last symbol of w. By assumption we have

$$\hat{\delta}_D(q_0, x) = \hat{\delta}_N(q_0, x)$$

$$= \{P_1, P_2, \cdots P_k\} \rightarrow \text{sub sets}$$

Now, by the definition of extended transition function of DFA,

$$\hat{\delta}_D(q_0, w) = \hat{\delta}_D(q_0, xa)$$

$$= \delta_D(\hat{\delta}_D(q_0, x), a)$$

$$= \delta_D(\{P_1, P_2 \cdots P_k\}, a)$$

$$= \delta_N(\hat{\delta}_N(q_0, x), a)$$

$$= \hat{\delta}_N(q_0, w).$$

Therefore, by induction principle, the theorem is true for all the strings w of the given language.

Thus, $L(D) = L(N)$.

NFA with ε transition. (ε-NFA)

* An interesting feature of NFA is that it allows ε transitions.

ε-transitions are defined as the transitions in which the NFA can change its state without reading anything from the input string.

* it is difficult to construct a DFA for a language that accepts all strings consisting of any number of zeros followed by any number of ones. But it is very easy to construct an ε-NFA for the same language.

The Formal notation of ε-NFA

An ε-NFA can be represented exactly as NFA, with one exception; the transition function must include information about transition on ε. We can represent ε-NFA

(27) $(Q, \Sigma, \delta, q_0, F)$ this.

Here $\delta : Q \times \Sigma \cup \{\varepsilon\} \rightarrow 2^Q$.

Eg:

suppose we have a language L
$$L = \{ a^n b^m c^w \mid n, m, w \geq 0 \}$$

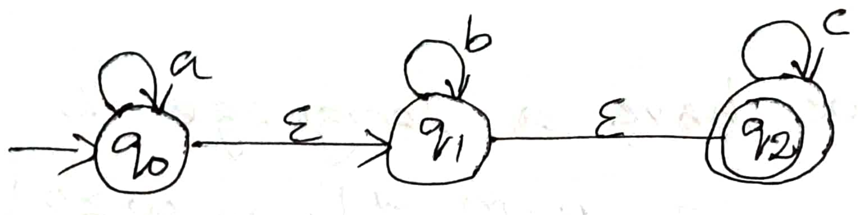
The strings Generated by this language can have any number of "a"s (including 0 as) and any number of "b"s follo-wed by any number of "c"s.
The one important thing here is the order of a, b, c. ∴ The only thing that needs to be taken care of is once we have accepted "b" after any number of "a"s, then a can not come. Similarly after receiving c both a and b can not come.

The possible strings are

$L = \{ a, b, c, aab, aac, bbc, abc, aabc, aabbc, bbcc, \ldots \}$.

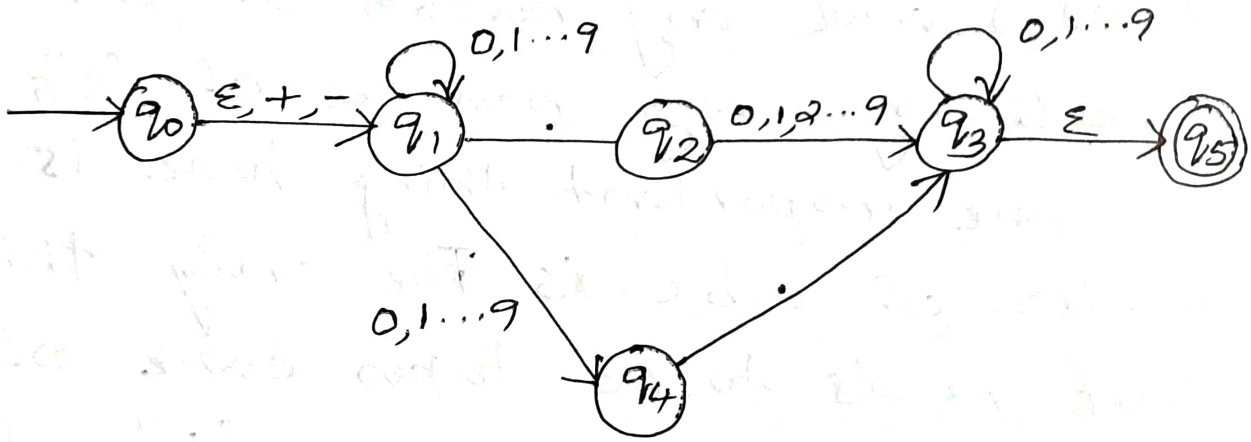
After b, a can not come. After c both a and b can not come.

So for a language like this, it is difficult to design a DFA, also an NFA. So we go for ε-NFA.



Eg 2:

ε-NFA for floating point numbers.



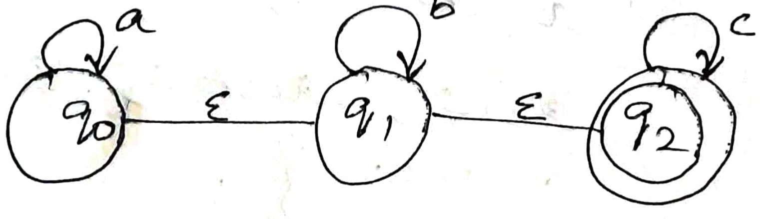
* ε-closure of a state.

ε-closure of a state q is defined as the set of all states which are reachable from q through ε.

ε-closure can be written as

ECLOSE( );

eg:



Here we can find the ECLOSE of

27

all the states.

$\hat{\delta}(q_0, \varepsilon)$ = ECLOSE (q_0)

= $\{q_0, q_1, q_2\}$.

Every state on ε goes to itself, even if the transition is not shown in the transition diagram.

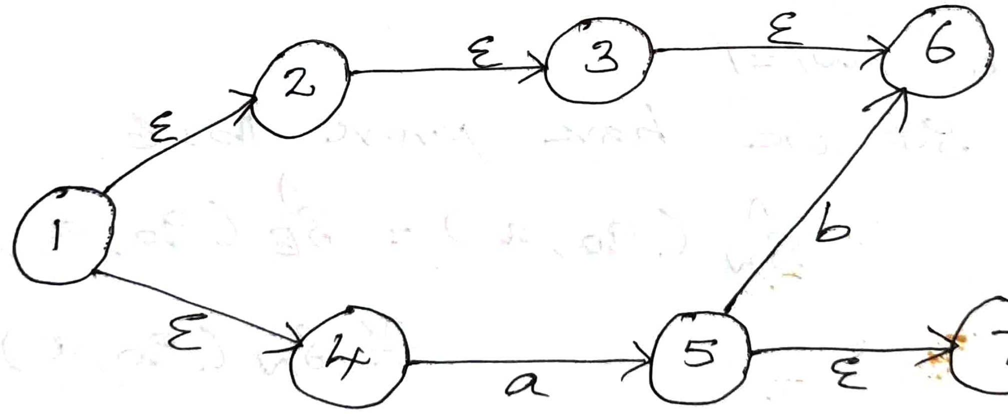
$\hat{\delta}(q_0, \varepsilon)$ = ECLOSE (q_1)

= $\{q_1, q_2\}$

$\hat{\delta}(q_2, \varepsilon)$ = ECLOSE (q_2)

= $\{q_2\}$.

another example.



Here ECLOSE (1) = $\{1, 2, 3, 6, 4\}$.

ECLOSE (2) = $\{2, 3, 6\}$

ECLOSE (4) = $\{4\}$

ECLOSE (3) = $\{3, 6\}$

etc...

Equivalence of E-NFA & NFA

Theorem:

Let $E = (Q, \Sigma, \delta_E, q_0, F)$ be an epsilon NFA and $N = (Q, \Sigma, \delta_N, q_0, F_N)$

where $F_N = \{ F \cup \{q_0\}$ if ECLOSE (q_0) contains a state in F

F otherwise $\}$ constructed from ϵ-NFA E

and we have to prove

that $L(E) = L(N)$.

Given that $\delta_N(q, a) = \hat{\delta_E}(q, a)$.

Basis of Induction:

Here $|w| = 1$

So we have prove that

$\hat{\delta_N}(q_0, a) = \hat{\delta_E}(q_0, a)$

$\stackrel{?}{=} \delta_N(q_0, a)$

[From the given statement $\delta_N(q_0, a) = \hat{\delta_E}(q, a)$]

Since "a" is a single character

$\hat{\delta_N}(q_0, a) = \delta_N(q_0, a)$

Hence the proof.

③ Inductive step (Hypothesis)

$$\hat{\delta}_N(q_0, x) = \hat{\delta}_E(q_0, x) = \{P_1, P_2 \cdots P_k\}$$

now we need to prove for $w = xa$

$$\hat{\delta}_N(q_0, w) = \hat{\delta}_E(q_0, w)$$

proof:

$$\hat{\delta}_N(q_0, w) = \hat{\delta}_N(\hat{\delta}(q_0, x), a)$$

$$= \hat{\delta}_N(\{P_1, P_2, \cdots P_k\}, a)$$

$$= \bigcup_{i=1}^{k} \hat{\delta}_N(P_i, a)$$

$$= \bigcup_{i=1}^{k} \delta_N(P_i, a)$$

Similarly

$$\hat{\delta}_E(q_0, w) = \hat{\delta}_E(\hat{\delta}_E(q_0, x), a)$$

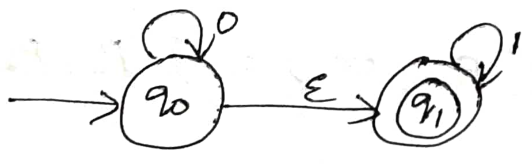
$$= \hat{\delta}_E(\{P_1, P_2 \cdots P_k\}, a)$$

$$= \bigcup_{i=1}^{k} \hat{\delta}_E(P_i, a)$$

$$= \bigcup_{i=1}^{k} \delta_N(P_i, a)$$

Hence proved.

32 *) ## ε-NFA to NFA conversion:



convert the given ε-NFA to NFA.

1) First find the ε-closure of all the states.

$$ECLOSE(q_0) = \{q_0, q_1\}$$

$$ECLOSE(q_1) = \{q_1\}$$

now we have to fill the transition table of NFA.

	0	1
q_0	q_0, q_1	q_1
q_1	ϕ	q_1

For filling the above table we need to find the epsilon closure of a state, from those states find the transition for a particular symbol (0/1) and then again find the epsilon closure of that states.

This can be done using a [illegible] table

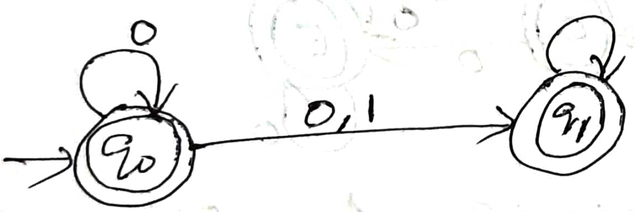
	ε^*	0	ε^*
q_0	q_0, q_1	q_0	q_0, q_1
q_1	q_1	ϕ	ϕ

This can be added to NFA $(q_0, 0)$ transition table entry.

$(q_1, 0)$

null set or blank space can be given.

	ε^*	1	ε^*
q_0	q_0, q_1	q_1	q_1
q_1	q_1	q_1	q_1



Here q_1 is the actual final state in ε-NFA. So we can make q_1 a final state in our NFA.
Apart from that, there is an ε transition from q_0 (a non final state) to q_1 (a final state). So q_0 will also be taken as final state.

MODULE II

REGULAR EXPRESSIONS:

The languages which are accepted by a Finite automata are known as Regular languages. The Grammar which is used to generate regular languages is known as Regular Grammar.

Regular Expressions (RE) are mathematical or algebraic notations used to represent regular languages.

Regular expressions are more user friendly to represent a language than FA.
For eg: 01^* is a regular expression denoting a language consists of all strings that are a single 0 followed by any number of 1s.

Formal definition:

A language $L(r)$ denoted by a regular expression r is defined by the following.

1. ϕ is a regular expression denoting the empty set $\{\}$.

2. ε is a regular expression denotes $\{\varepsilon\}$

3. For each a in Σ, 'a' is a regular expression.

The above 3 are basic or primitive regular Expressions.
Mainly 3 operations can be applied on primitive REs and then we can get other regular Expressions.
The main 3 operations are

1. union (denoted by "+")

$r_1 \rightarrow$ a (basic) RE

$r_2 \rightarrow$ a (basic) RE

then $r_1 + r_2$ is also a regular expression indicating that either r_1 or r_2. This means that the collection of strings present either in r_1 or in r_2.

Eg: $01^* + 10^* \rightarrow$ indicates strings that are a single zero followed by any number of 1s or a single 1 followed by any number of 0s.

2. concatenation. (denoted by ".")

$r_1 \rightarrow$ a RE

$r_2 \rightarrow$ a RE

then $r_1 \cdot r_2$ is also a regular expression indicating that r_1 is followed by r_2. This means that the collection of strings where each string is a combination of the string from r_1 and r_2.

3. CLOSURE

1. Kleene closure: (denoted by "*")

$r_1 \rightarrow$ a RE

thus r_1^* is also a regular Expression defined as any number of occurrences of r_1 including zero in other words we can say that "0 or more occurrences

2. positive closure (+)

$r_1 \rightarrow$ a RE

thus r_1^+ is also a regular Expression. It is defined as one or more occurrences of r_1

$$r_1^* = r_1^+ \cup \varepsilon .$$

r_1^+ does not include "ε", but r_1^* includes "ε".

Examples.

1) write a RE for a language accepting strings of a's and b's of any length including null string (ε).

$\Sigma = \{a, b\}$ $L = \{\varepsilon, a, b, aa, ab \ldots\}$

The RE $= (a+b)^*$

2) write a RE for a language accepting strings of length exactly 2 over $\Sigma = \{a, b\}$

20

Ans) $L = \{aa, ab, ba, bb\}$

Since this language is finite we can perform union of all the strings

$$RE \Rightarrow aa + ab + ba + bb$$

$$\Rightarrow a(a+b) + b(a+b)$$

$$\Rightarrow (a+b)(a+b)$$

So answer is $(a+b)(a+b)$

3) RE for a language accepting strings of length atleast 2.

$L = \{aa, ab, ba, bb, aab, aba, baa, bba, \ldots\}$

Here the language is not finite.

So we can not perform union

we already have regular expression for strings of length exactly 2. So, we can make our regular expression from that considering it as a base.

So our answer is $\underline{(a+b)(a+b)(a+b)^*}$

4) RE for a language accepting strings of length atmost 2. $\Sigma = \{a, b\}$

$L = \{\varepsilon, a, b, aa, ab, ba, bb\}$

$RE \rightarrow \varepsilon + a + b + aa + ab + ba + bb$

$\Rightarrow (\varepsilon + a + b)(\varepsilon + a + b)$

5) write a RE for the language accepting the strings which are starting with 1, and ending with 0 over the set $\Sigma = \{0, 1\}$

$L = \{10, 100, 110, 1010, 1110, 1000 \ldots\}$

RE $\Rightarrow$ 1 (any number / combination of 0s and 1s) 0

$\Rightarrow 1(0+1)^* 0$

6) RE for a language accepting all combination of 0s and 1s ending with 00 over $\Sigma = \{0, 1\}$

RE $\Rightarrow$ (any combination of 0s & 1s) 00

$\Rightarrow (0+1)^* 00$

* RE for a language accepting strings that contain the substring ab

RE $\Rightarrow$ (any combination of a's & bs) ab (any comb of a's & bs)

$\Rightarrow (a+b)^* ab (a+b)^*$

(22) 7) Write the regular expression for a language over $\Sigma = \{a, b\}$ such that all the strings do not contain the substring "ab".

$L = \{\varepsilon, a, b, aa, bb, ba, aaa \ldots\}$

R.E = $b^* a^*$

* REGULAR EXPRESSION TO FINITE AUTOMATA.

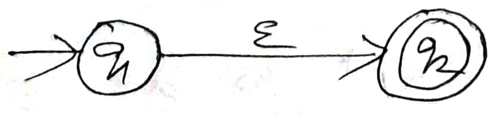
A regular expression can be converted to a finite automata (either dfa, nfa or ε-nfa).

Here we use Thompsons method to construct an ε-nfa from a regular expression.

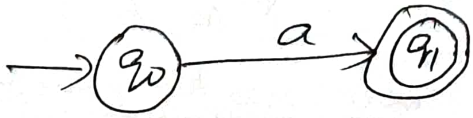
So the base for creating an ε-nfa from RE using thompsons method are the following things.

1. For a RE ϕ our ε-nfa or FA is

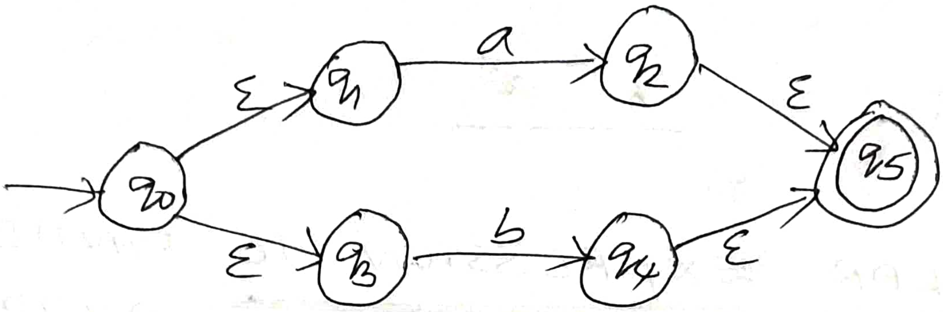
2. For a RE ε



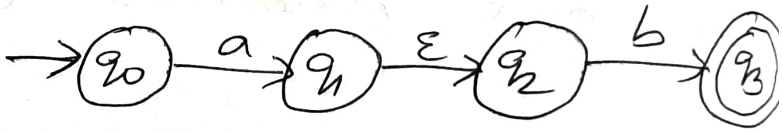
3. For a RE "a"



4. For a RE "a+b"

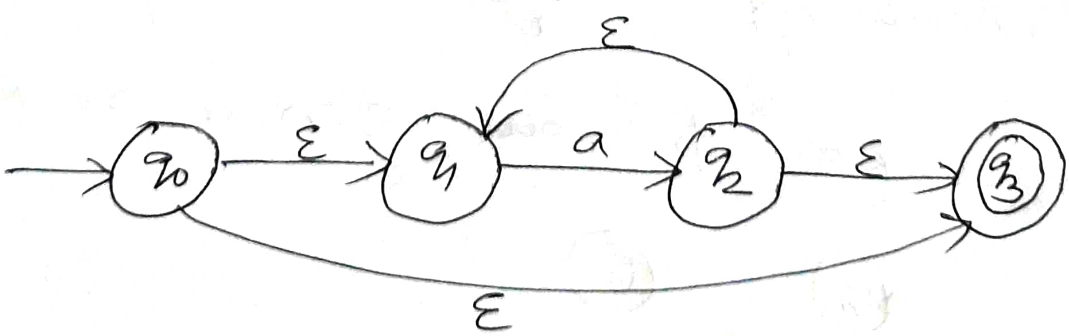


5. For a RE "ab"

it can be 

or

6. For a RE "a^*"

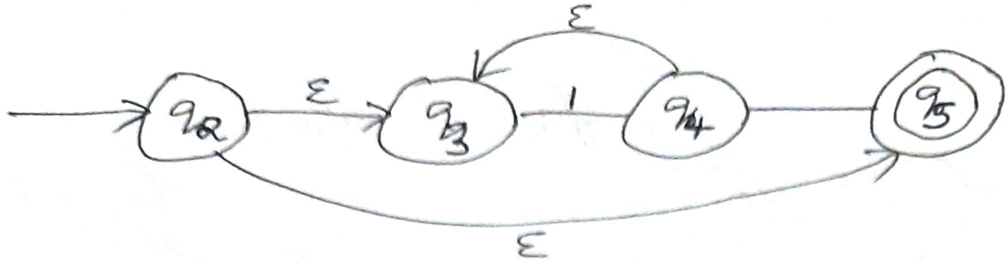


Eg:

1) Convert a RE 01^* to an ε-nfa.

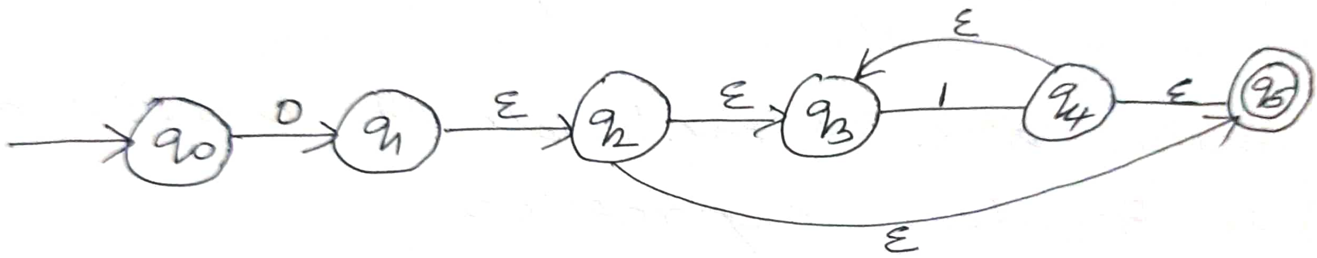
24 This can be done in 2 steps.

First create an ε-NFA for 1^*.



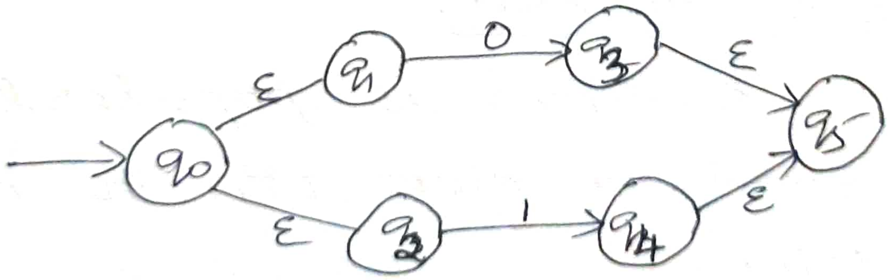
now add "0" to the above created ε-NFA in the front

[0 is concatenated with 1^*]

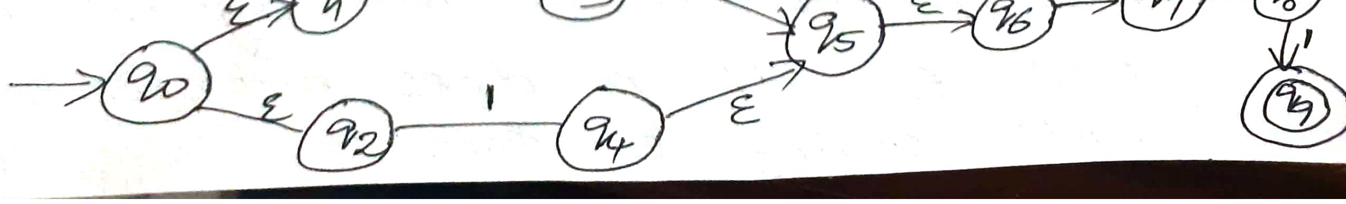


2) convert the regular expression $(0+1)01$ to an ε-NFA

Step 1: draw the RE for $(0+1)$

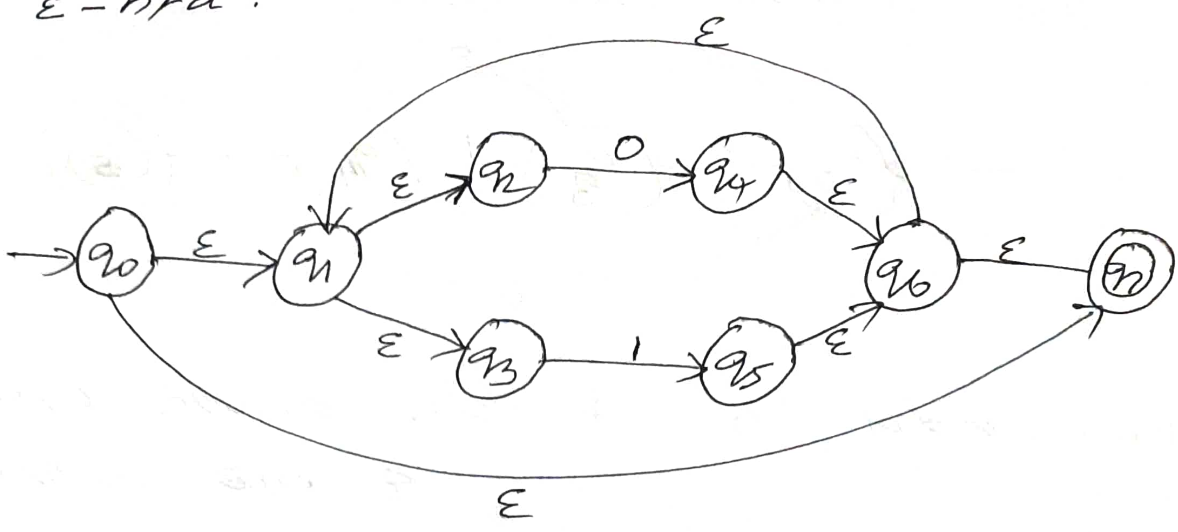


now concatenate 01 to the end.



(25)

3) convert the regular expression $(0+1)^*$ to an ε-nfa.



* DFA to Regular Expression

There are 2 methods to convert a DFA to Regular expression.

1) Arden's method.
2) state elimination method.

Here we discuss Arden's method.

Arden's Method.

Arden's theorem is popularly used to convert a given DFA to its regular expression.

it states that -

Let P and Q be two regular expression over Σ. if P does not contain a null string ε, then

$R = Q + RP$ has a unique solution

$R = QP^*$

To use Arden's method, Following conditions must be satisfied.

1) The transition diagram must not have ε transitions.

2) There must be only a single initial state.

The Following steps are there to convert a DFA to RE.

1) Form an equation for each state considering the transitions which comes towards that state.

2) Add "ε" in the equation of initial state.

3) Bring final state in the form $R = Q + RP$ to get the required expression.

* Ardens theorem can be used for both NFA & DFA to a regular expression.

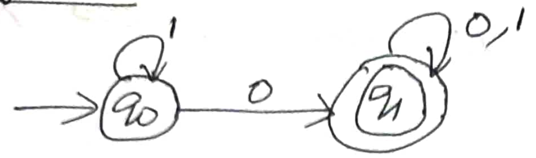
* if there are multiple final states then

1) write a RE for each final state

separately.

2) Add all the regular expression to get the final regular expression.

Eg:



now write the equation for final state

q_1:

$q_1 = q_0 0 + q_1 0 + q_1 1$

$\Rightarrow q_0 0 + q_1 (0+1)$ —— ①.

now we have to write the equation for q_0 to solve ①

$q_0 = \varepsilon + q_0 1$ ("ε" is added here because q_0 is initial state).

→ This is in the form of $R = Q + RP$

$Q = \varepsilon$

$P = 1$.

So our solution can be written as

$q_0 = \varepsilon 1^*$

$= 1^*$

now apply the value of q_0 in ①.

(28)

(1) becomes

$$q_1 = 1^*0 + q_1(0+1)$$

now this is also in the form
$R = Q + RP$.

$$q_1 = 1^*0 \cdot (0+1)^*$$

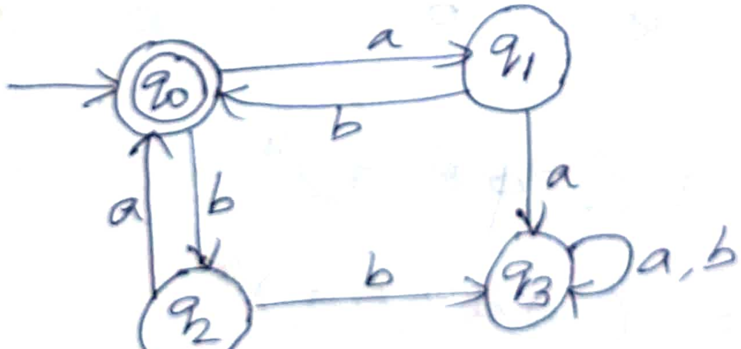
here $Q = 1^*0$
$P = (0+1)$

So by solving the equation correspon-
-ding to the final state (Here q_1)
we get our resultant RE

$$\underline{R.E = 1^*0 \cdot (0+1)^*}$$

$$RE = 1^*0(0+1)^*$$

Eg: 2.



convert this DFA to a regular Expression.

*) write the equation for final state.
Here final state is q_0, (which is the initial state too).
So when writing the equation for initial state we have to include "ε"

29

$q_0 = q_2 a + q_1 b + \varepsilon$ —— ①

now we have to write the equations for q_1 and q_2.

$q_1 = q_0 a$ —— ②

$q_2 = q_0 b.$ —— ③

apply ② and ③ in ①

$$q_0 = q_0 ba + q_0 ab + \varepsilon$$

$$= \varepsilon + q_0 (ba + ab)$$

$$\underset{R}{q_0} = \underset{Q}{\varepsilon} + \underset{R}{q_0} \underset{P}{(ab + ba)}$$

$R = Q + RP$

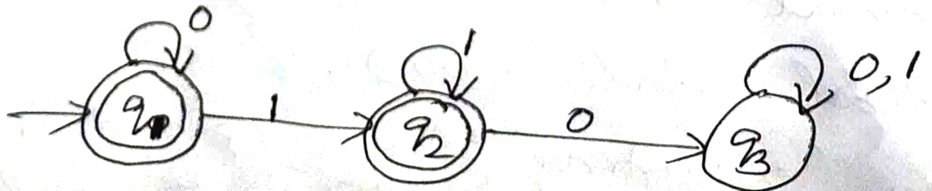
$R = QP^*$

$$\text{so } q_0 = \varepsilon (ab + ba)^*$$

$$= (ab + ba)^*$$

$$\text{so } RE = (ab + ba)^*$$

Eg: 3



(3)

Here our initial state → q_1.
we have 2 final states here q_1, q_2.
So we have to find the RE for each final state and thus add those REs

*) write the equation for q_1 (which is initial state (include ε) as well as final state.

$$q_1 = \varepsilon + q_1 0 \quad \text{—①} \quad [R = Q + RP]$$

$$q_1 = \varepsilon \cdot 0^* \qquad R = q_1$$
$$Q = \varepsilon$$
$$= 0^* \qquad P = 0$$

Now find the equation for q_2.

$$q_2 = q_1 1 + q_2 1 \quad \text{—— ②}$$

now apply the value of q_1 (0^*) in ②

$$\underset{R}{q_2} = \underset{Q}{0^* 1} + \underset{R}{q_2}\,\underset{P}{1} \qquad [R = Q + RP]$$

$$q_2 = 0^* 1 1^*$$
$$= 0^* 1^+ \qquad [1 1^* = 1^+]$$

(31) Now add the regular Expression of q_1 and q_2.

$$q_1 = 0^*$$

$$q_2 = 0^* 1^+$$

So our $RE = q_1 + q_2$

$$= 0^* + 0^* 1^+$$

REGULAR GRAMMAR

A regular grammar is one in which any production contains only one Non-terminal at the LHS and the RHS contains only one terminal

$A \rightarrow \epsilon$

$A \rightarrow a$

$A \rightarrow aB$

A regular grammar is defined as

$$G = (V, T, P, S)$$

$V \Rightarrow$ set of non-terminals (N can also be used)

$T \Rightarrow$ set of terminals

$P \Rightarrow$ productions.

$S \Rightarrow$ start symbol $\in V$.

There are 2 types of regular grammar

1) Right-linear grammar

2) left linear grammar

A grammar is said to be right linear if all the production rules are the form

$$A \rightarrow aB$$
$$A \rightarrow a$$

A grammar is said to be left linear if all the productions are the form

$$A \rightarrow Ba$$
$$A \rightarrow a.$$

* EQUIVALENCE OF REGULAR EXPRESSION & FINITE AUTOMATA.

Theorem:

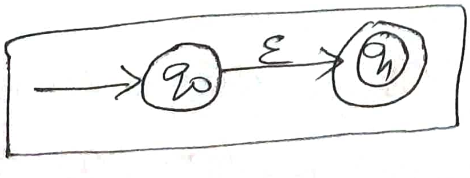
Given a regular expression r, there exists an ϵ-NFA M such that M accepts $L(R)$.

ie, $L(M) = L(r)$

To prove this theorem, mathematical induction is used on the number of operators in r.

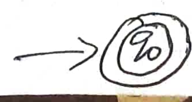
Base: (no. of. operators = 0)

1) $\epsilon \longrightarrow L(\epsilon) = \{\epsilon\}$

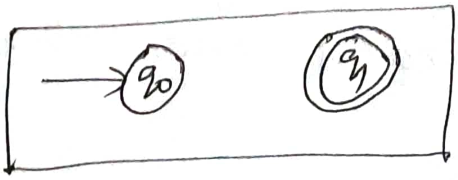


M $\quad L(M) = \{\epsilon\}$

or

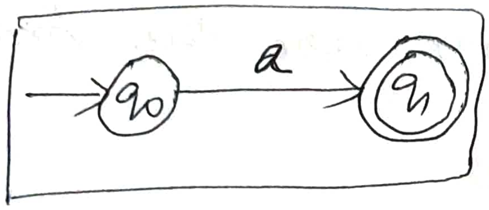


2) ϕ, $L(\phi) = \phi$.



M $\quad L(M) = \phi$.

3) a, $L(a) = a$ $[a \in \Sigma]$



M $\quad L(M) = \{a\}$

The Statement (theorem) is true for base case.

Inductive step. (1 or more number of operators).

Induction hypothesis: Assume the theorem is true for a regular expression "r" with fewer than i ($i \geq 1$) operators.
(That means theorem is true for $k < i$)

Now we have to prove that theorem is true for $k = i$

Let "r" be any regular expression with i (number) operators.

case 1: $r = r_1 + r_2$, no. of operators in $r_1, r_2 < i$.

case 2 : $r = r_1 \cdot r_2$, no. of operators in $r_1, r_2 < i$

case 3 : $r = r_1^*$, no. of operators in $r_1 < i$.

case 1:

$r = r_1 + r_2$.

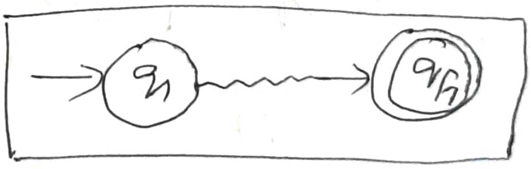
$\to M_2$ (ε-NFA)

$\to M_1$ (ε-NFA)

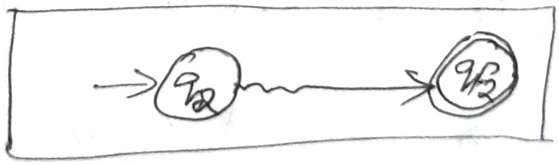
$L(M_1) = L(r_1)$

$L(M_2) = L(r_2)$. by Induction Hypothesis.

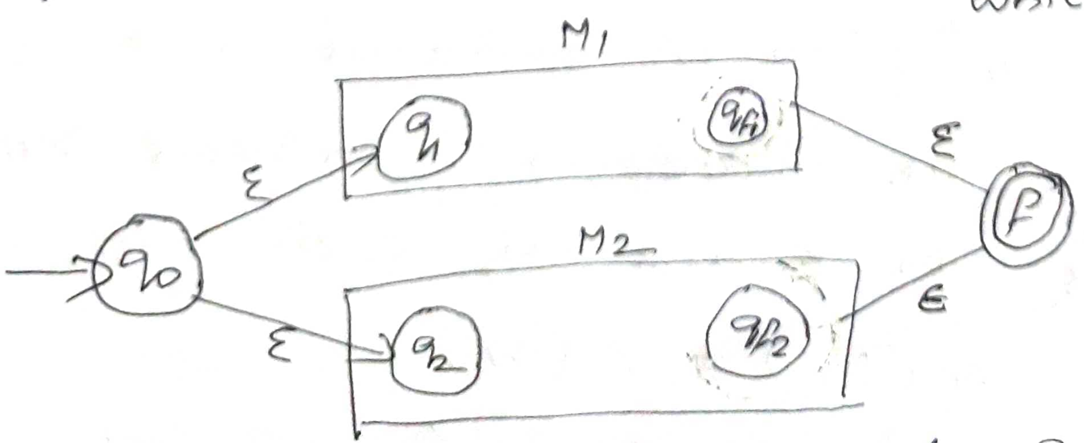
Suppose $M_1 :\Rightarrow$



$M_2 : \Rightarrow$



Now we can construct a new M which accepts "r"



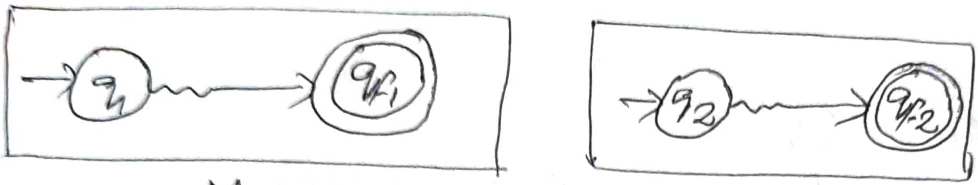
$L(M) = L(r_1) \cup L(r_2)$.

From this we can say

$$r = r_1 + r_2 .$$

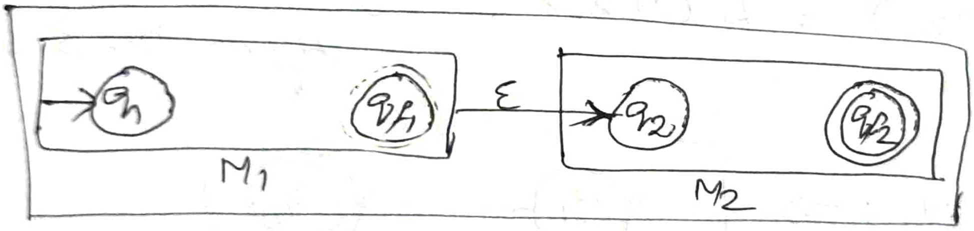
case 2:

$$r = r_1 \cdot r_2 .$$



M_1 M_2

From this we can construct our new M



M

$$L(M) = L(r_1) \cdot L(r_2)$$

$$r = r_1 \cdot r_2 .$$

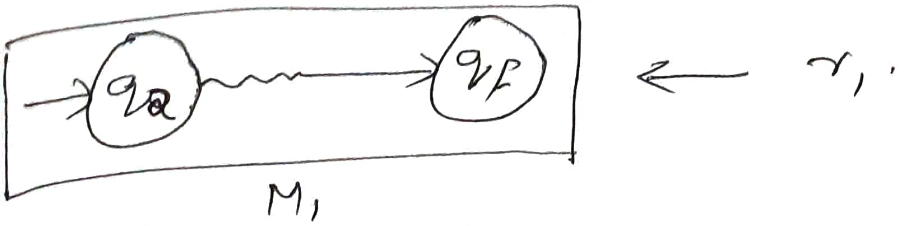
case 3:

$r = r_1^*$ [no. of. operators in $r_1 < i$]

so by the induction hypothesis there exists an ε-nfa M_1 such that

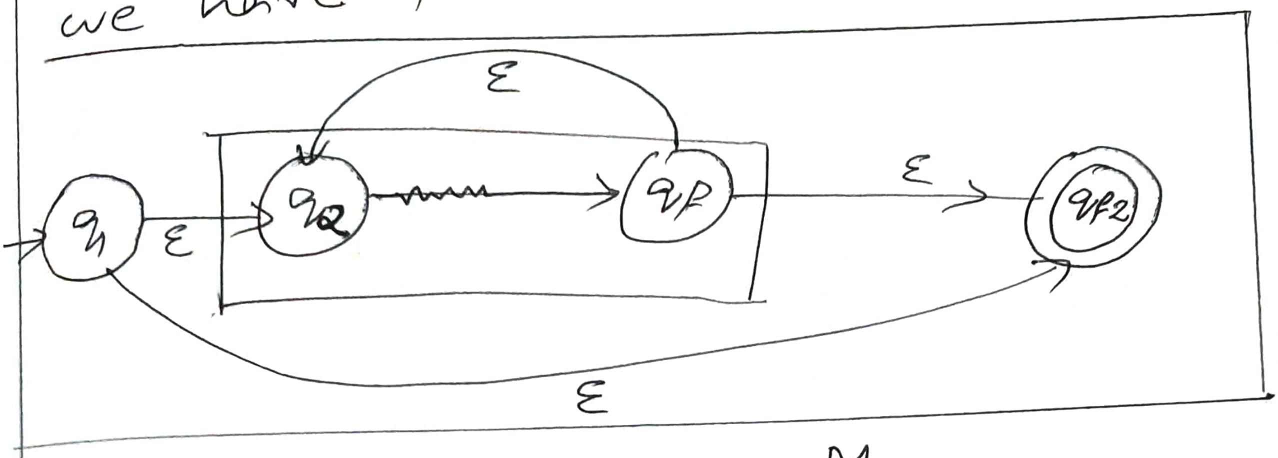
$$L(M_1) = L(r_1)$$

we have to prove that we can construct an ε-nfa for "r" [r_1^*]



M_1

we have to construct $M(r_1^*)$



M

$$L(M) = (L(r_1))^*$$
$$= L(r).$$

so we have proved all the three cases for "i" no. of operators. Hence the theorem is proved.

Finite Automata to Regular Grammar

Let $M = (\{q_0, q_1 \ldots q_n\}, \Sigma, \delta, q_0, F)$

be a DFA

The equivalent grammar G can be constructed from this DFA.

$$G = (\{A_0, A_1, A_2 \ldots A_n\}, \Sigma, P, A_0).$$

The set of production rules P can be defined by the following rules.

1) if $\delta(q_i, a) = q_j$, where $q_j \notin F$.

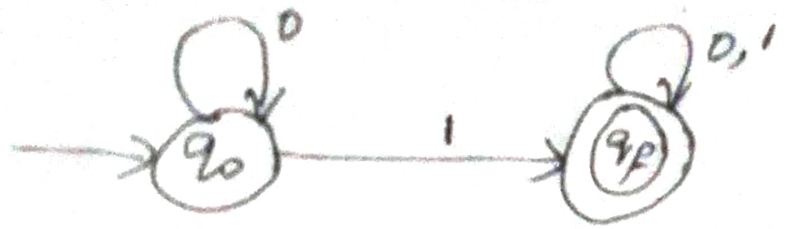
then $A_i \rightarrow a A_j$

2) if $\delta(q_i, a) = q_j$ where $q_j \in F$

then $A_i \rightarrow a A_j$

and $A_i \rightarrow a$

eg:



$G = (V, T, P, S)$

$V = \{A_0, A_1\}$ $\quad q_0 = A_0$

$T = \{0, 1\}$ $\quad q_f = A_1$.

$A_0 \rightarrow 0A_0 \mid 1A_1 \mid 1$

$A_1 \rightarrow 0A_1 \mid 1A_1 \mid 0 \mid 1$

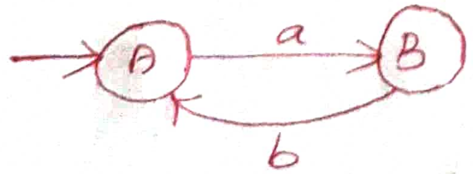
1

DFA to RE using State Elimination method.

steps:

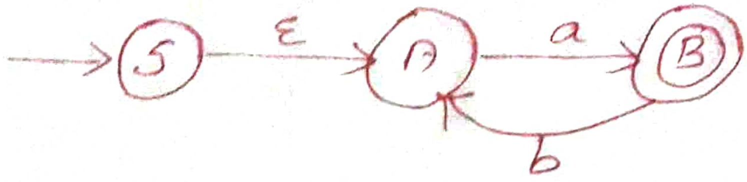
1) If there exists any incoming edge to initial state, create a new initial state having no incoming edge.
2) In case of multiple final states, convert them into non-final states and create a new final state.
3) If there exists any outgoing edge from final state, create a new final state having no outgoing edge.
4) Eliminate all intermediate states one by one. only initial and final states should be there.

Step 1 & 2

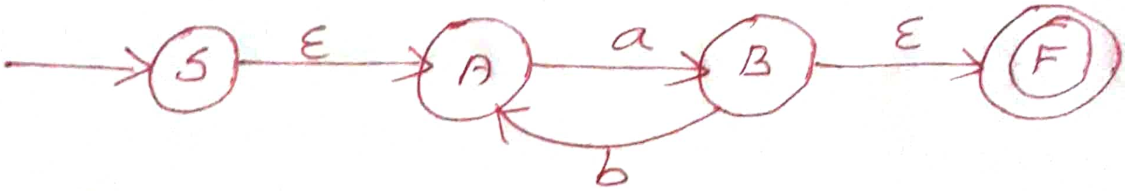


Here the initial state "A" has an incoming edge. So we have to create a new initial state.

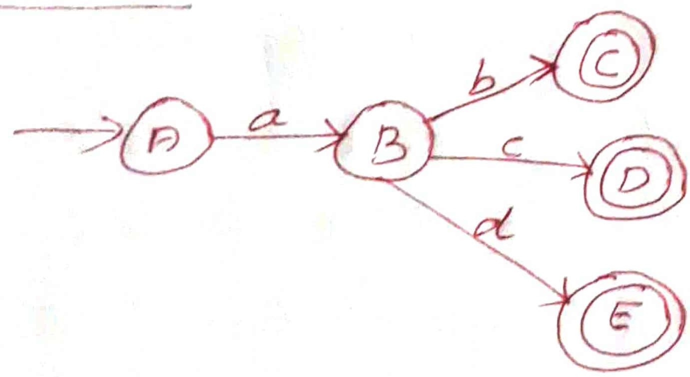
2



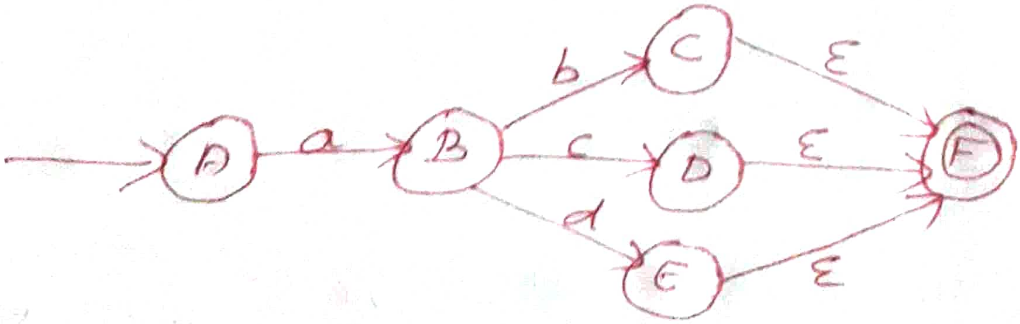
Here the final state "B" has an out-going edge. So we have to create a new final state.



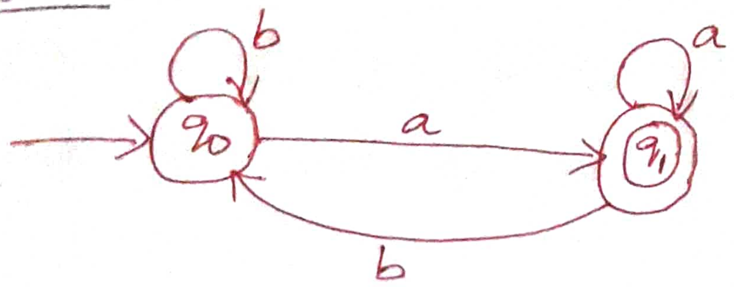
step 3:



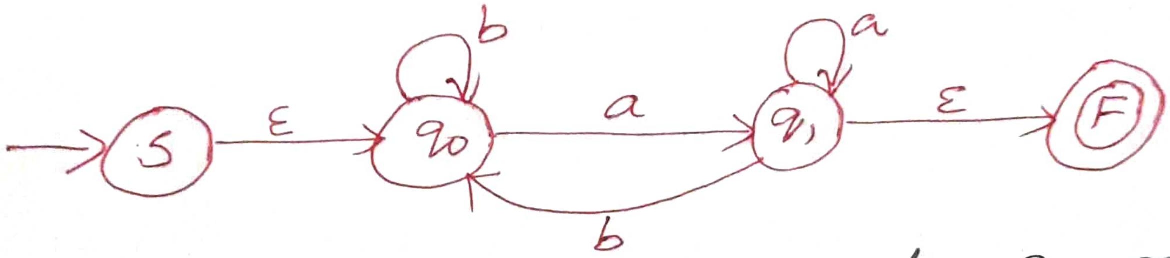
Here we have 3 final states. So we have to make these states as non-final state and create a new final state.



Eg 1:



Here rule 1 and 2 are violated. So we have to create a new initial and final states by using incoming ε transition and outgoing ε - transition.



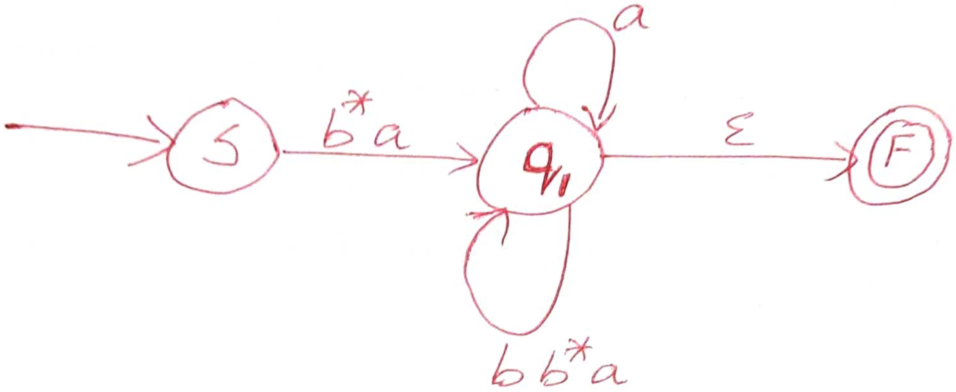
Now we have to eliminate q_0 and q_1 in any order as we wish. (The answer will be same).

Lets eliminate "q_0" first.

To eliminate q_0, Imagine if "q_0" was not there we can reach "q_1" from "S" using the input $\varepsilon b^* a$, neglect "ε" so the input will be $b^* a$. from S to q_1

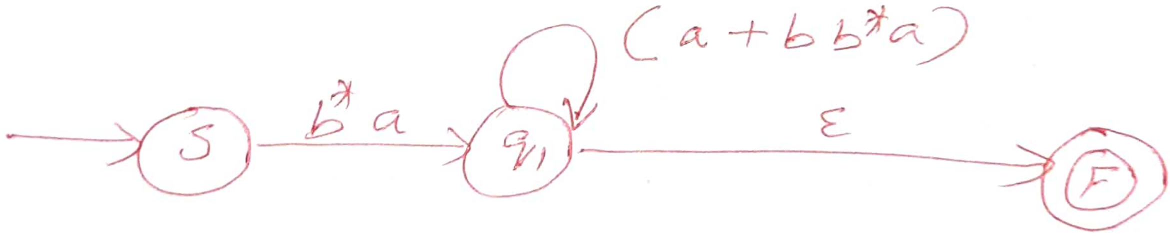
In this case there is an outgoing edge from "q_1" to "q_0". when "q_0" is eliminated, that edge becomes a self loop to "q_1".

Then, there are 2 self loops for q_1. one is "bb^*a" (after the removal of q_0) and the other one is the already existing one "a".

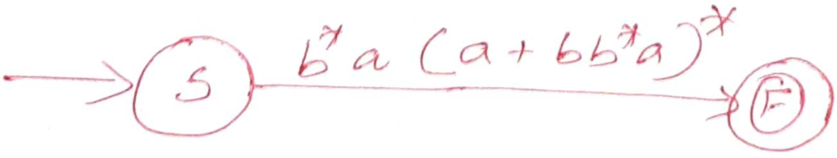


Now, we eliminate q_1.

If we eliminate q_1 thus we can reach the final state "F" directly from S.



after eliminating q_1.



So our RE: $b^*a(a+bb^*a)^*$

Module - 2

Minimization of DFA (Equivalence partition)

Steps:

1) Divide Q (set of states) in to 2 sets. one set will contain all final states and other will contain all non final states. This partition is called P_0.

2) initialize $K=1$.

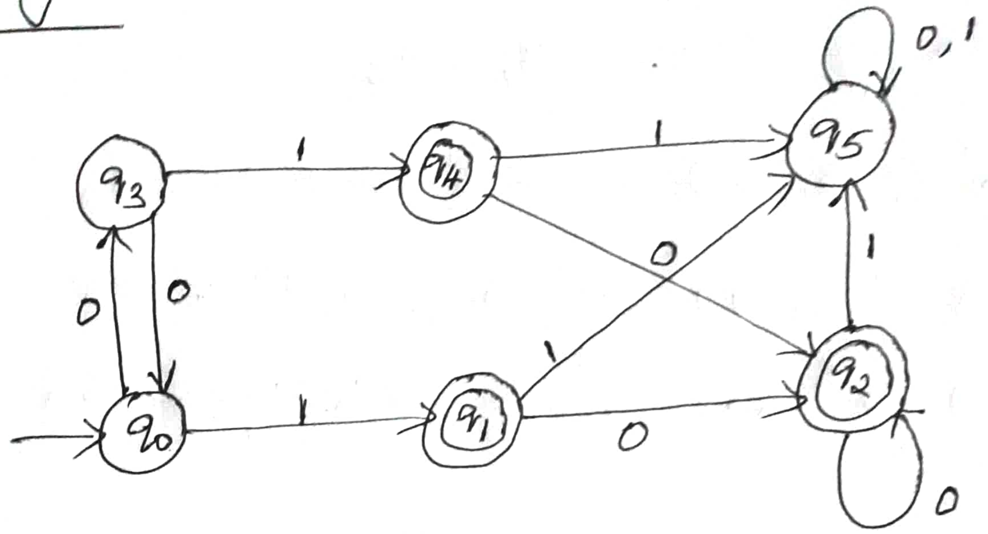
3) Find P_K by partitioning different sets of P_{K-1}. In each set of P_{K-1}, we will take all possible pair of states. If two ~~sets~~ states of a set are distinguishable, we will split the set into different sets in P_K.

4) Stop when $P_K = P_{K-1}$ (No change in partition)

5) All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of states in P_K.

*) Two states q_i, q_j are distinguish-able in partition P_k if for any input symbol a, $\delta(q_i, a)$ and $\delta(q_j, a)$ are in different sets in partition P_{k-1}.

Eg:



$P_0 = \{ \{q_1, q_2, q_4\} \{q_0, q_3, q_5\} \}$

P_0 has 2 different set of states.

Now we have to check whether P_0 can be partitioned to get P_1.

First consider the set
$\{q_1, q_2, q_4\}$.

Take the pair of states q_1, q_2

$\delta(q_1, 0) = \delta(q_2, 0) = q_2$.

$\delta(q_1, 1) =$

$\delta(q_2, 1) = q_5$

So q_1, q_2 are not distinguishable

	0	1
q_0	q_3	q_1
q_1	q_2	q_5
q_2	q_2	q_5
q_3	q_0	q_4
q_4	q_2	q_5
q_5	q_5	q_5

Similarly take q_1, q_4

$$\delta(q_1, 0) = \delta(q_4, 0) = q_2$$

$$\delta(q_1, 1) = \delta(q_4, 1) = q_5$$

So they are not distinguishable.

Since q_1, q_2 and q_1, q_4 are not distinguishable, q_2, q_4 are not distinguishable. So $\{q_1, q_2, q_4\}$ will not be partitioned.

Consider the set $\{q_0, q_3, q_5\}$

$\delta(q_0, 0) = q_3$ and $\delta(q_3, 0) = q_0$.

$\delta(q_0, 1) = q_1$ and $\delta(q_3, 1) = q_4$.

Here q_0, q_3 are not distinguishable.

Now consider q_0, q_5

$\delta(q_0, 0) = q_3$ $\delta(q_5, 0) = q_5$

$\delta(q_0, 1) = q_1$, $\delta(q_5, 1) = q_5$

The moves of q_0 and q_5 on input symbol "1" are q_1 and q_5 which are in different set in partition P_0.
Hence, q_0 and q_5 are distinguishable.

So $\{q_0, q_3, q_5\}$ will be partitioned into

$\{q_0, q_3\}$ and $\{q_5\}$

So $P_1 = \{\{q_1, q_2, q_4\}, \{q_0, q_3\}, \{q_5\}\}$

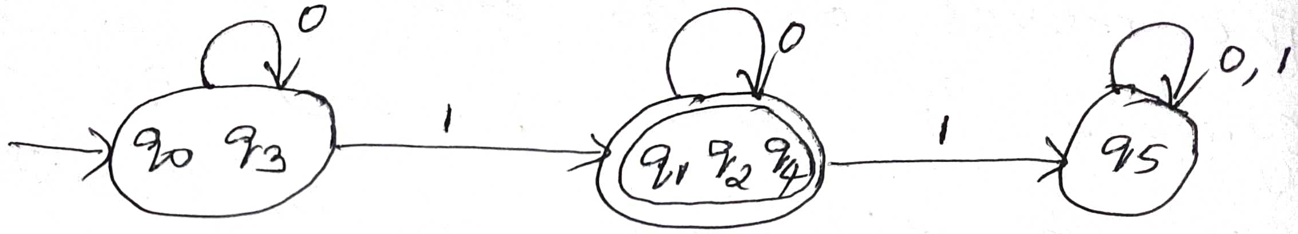
To calculate P_2, we will check whether sets of partition P_1 can be partitioned or not.
In this example we get

$$P_2 = \{\{q_1, q_2, q_4\}, \{q_0, q_3\}, \{q_5\}\}$$

which is same as P_1.

So here the sets $\{q_0, q_2, q_4\}$ are merged into one state, similarly $\{q_0, q_3\}$ are merged into one state, and q_5 is taken as another set.

So the minimized DFA is



Pumping Lemma for Regular Language

For any regular language L, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists $u, v, w \in \Sigma^*$, such that $x = uvw$

and

1) $|uv| \leq n$
2) $|v| \geq 1$
3) for all $i \geq 0$, $uv^iw \in L$

In simple terms this means if a string 'v' is pumped ie, if "v" is inserted any number of times, the resultant string still remains in L.

pumping lemma is used to prove a language is not regular. So if a language does not satisfy the conditions of pumping lemma then that means that language is not regular. The opposite of this may not always be true. That means if pumping lemma holds, it does not mean the language is regular.

Eg:

$L = \{a^n b^n \mid n \geq 0\}$ prove that this language is not regular / check whether the given language is regular or not.

$L = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$.

now we have to consider our pumping length (n). Here we can take any number. Suppose $n = 2$.

consider a string aabb

now split the string into xyz such that
1) $|xy| \leq n$
2) $|y| \geq 1$

$x \rightarrow a$
$y \rightarrow a$
$z \rightarrow bb$

now we have to check $xy^iz \in L$ for all $i \geq 0$

when $i=1 \Rightarrow a\,a^1\,bb$
$\Rightarrow aabb \in L$

when $i=2 \Rightarrow a\,a^2\,bb$
$\Rightarrow aaabb$ This does not belong to L

Since $aaabb \notin L$ we can say that

③ The third condition of the pumping lemma fails and this language is not regular.

Eg 2:

prove that $L = \{ a^i \mid i$ is prime numbers $\}$.

$L = \{ aa, aaa, aaaaa, aaaaaaa, \}$

Lets take our pumping length

$n = 3$

consider a given string in the language

$x = aaa$

now divide the string into 3 parts

$u = a$ 1) $|uv| \leq n$

$v = a$

$w = a$ 2) $|v| \geq 1$

So both 1 and 2 are satisfied.

Now we have to check the third condition

$uv^iw \in L$, for $i \geq 0$

$i = 1 \Rightarrow aa^1a$

$\Rightarrow aaa \in L$

$i = 2 \Rightarrow aa^2a$

$\Rightarrow aaaa \notin L$

Now the third condition fails for this particular string.
Now we can say that this language is not regular.
Hence proved.

* Applications of pumping Lemma

pumping lemma is used to prove that certain languages are not regular. It should never be used to show a language is regular.

ie 1) if L is regular, it satisfies pumping lemma

2) if L does not satisfy pumping Lemma, it is non regular.

* closure properties of Regular sets.

* if certain languages are regular and a language is formed from them by certain operations then L is also regular.

closure properties:

1) closure under union:

The union of two regular sets is regular.

ie, if L_1 and L_2 are 2 regular sets

then $L_1 \cup L_2$ is also regular.

eg: $L_1 = \{ a, aaa, aaaaa, \dots \}$

[odd length]

$L_2 = \{ \varepsilon, aa, aaaa, \dots \}$ [even length]

$L_1 \cup L_2 = \{ \varepsilon, a, aa, aaa \dots \}$

$L_1 \cup L_2$ is also regular.

2) closure under intersection.

The intersection of 2 regular languages (sets) is also regular.

ie, if L_1 and L_2 are 2 regular sets

then $L_1 \cap L_2$ is also regular.

3) closure under complementation.

The complement of a regular set is regular.

ie, if L_1 is a regular set then L_1' is also regular.

4) closure under difference

The difference of 2 regular languages is regular ie, if L_1 and L_2 are 2 regular sets then $L_1 - L_2$ is also regular.

5) closure under reverse.

The reversal of a regular set is regular if L is a regular set then L^R is also regular.

6) closure:

The closure (Kleene *) of a regular set is regular.
if L is a regular set then L^* is also regular.

7) closure under concatenation:

The concatenation of a regular language is regular.
if L_1 and L_2 are 2 regular sets, then $L_1 L_2$ is regular.

8) closure under Homomorphism:

A homomorphism (substitution of strings for symbols) of a regular set is regular.

9) closure under inverse Homomorphism

The inverse Homomorphism of a regular set is regular.

Homomorphism in Regular Languages:

Homomorphism of a language is represented by $h(L)$.

$$h(L) = \{ h(w) \mid w \text{ belongs to } L \}$$

$h : \Sigma$ to Γ^* (Mapping from Σ to Γ).

is called homomorphism.

eg: $\Sigma = \{0, 1\}$ $\Gamma = \{a, b\}$

$h(0) = aa$ $h(1) = bb$.

if $L = \{00, 101\}$

$h(L) = \{aaaa, bbaabb\}$

Similarly for a RE

$$(0+1)^* 1^*$$

$$\Downarrow$$

$$(aa+bb)^*(bb)^*$$

MODULE III

① Module - 3 (part - 1)

* Myhill - Nerode relations :

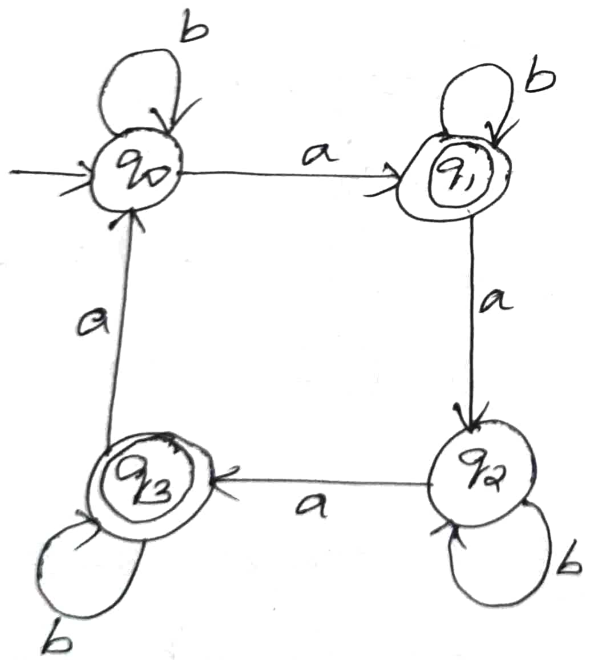
Consider a binary relation $\equiv_A$ on Σ^* induced by $A = (Q, s, \delta, F)$ on Σ defined as

$$\forall x, y \in \Sigma^*,$$

$$x \equiv_A y \iff \hat{\delta}(s, x) = \hat{\delta}(s, y)$$

[The processing of x & y from s will end in the same state]

Consider an example, DFA A.



eg : $(\varepsilon, b) \in \equiv_A$. This is because

$$\hat{\delta}(q_0, \varepsilon) = q_0$$

$$\hat{\delta}(q_0, b) = q_0$$

②

So for both these strings ε and b the processing is ended in the same state q_0. So these strings are related.

other example: $(\varepsilon, a^4) \in \equiv_A$

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

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

other examples $(a, ba) \in \equiv_A$.

$(a, aba^4) \in \equiv_A$.

(aa, aba)

$\Downarrow$

$$\hat{\delta}(q_0, aa) = q_2$$

$$\hat{\delta}(q_0, aba) = q_2$$

Now we are going to prove that it is [above binary relation] is an

1) $\equiv_A$ is an Equivalence relation.

a) Reflexive : $\forall x \in \Sigma^*$

$$\hat{\delta}(s, x) = \hat{\delta}(s, x)$$

b) Symmetric:

$$x \equiv_A y \implies \hat{\delta}(s,x) = \hat{\delta}(s,y)$$

$$\implies \hat{\delta}(s,y) = \hat{\delta}(s,x)$$

$$\implies y \equiv_A x.$$

c) Transitive:

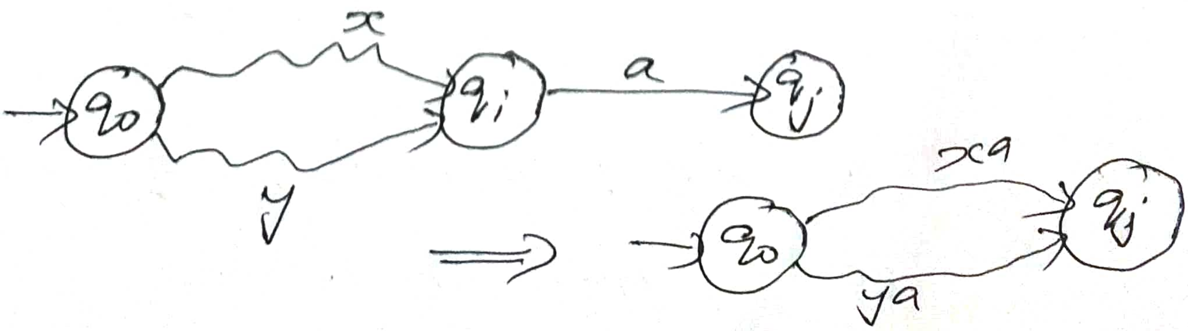
Suppose $x \equiv_A y$ and $y \equiv_A z$

$$\implies x \equiv_A z.$$

So the relation induced by the DFA A. over the set of ~~alphabets~~ all strings (Σ^*) is Reflexive, symmetric, Transitive. So this relation is an equivalence relation.

2) $\equiv_A$ is a right congruence:

$$\forall a \in \Sigma,\ x \equiv_A y \implies xa \equiv_A ya.$$

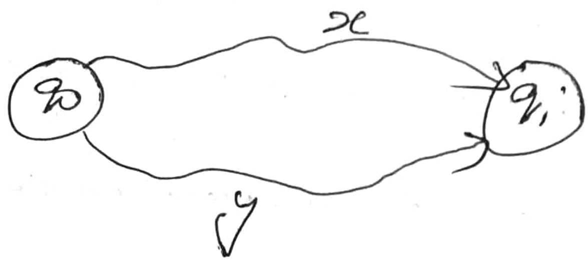


(4) iii) $\equiv_A$ refines $L(A)$.

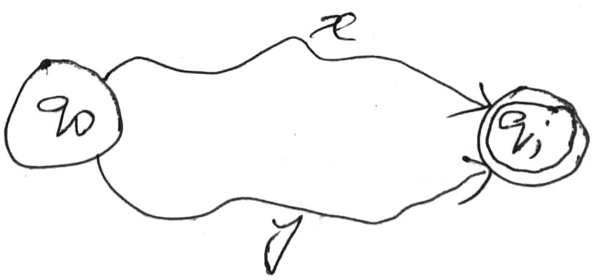
ie, whenever x and y related either x and y are in the Language or. they are not in the Language.

$$(x \equiv_A y \Rightarrow (x \in L(A) \Leftrightarrow y \in L(A))$$

Suppose $x \equiv_A y$, Then the paths for x and y goes to the same state



Now, if $x \in L(A)$, then q_i must be final state.



Here 'y' is also accepted. So $y \in L(A)$.

Now, if $x \notin L(A)$, then q_i must be a non final state

(5) thus $y \notin L(A)$.

IV) $\equiv_A$ is of finite index ie,
there are finitely many equivalence classes.

so for this particular string there are mainly 4 equivalence classes.

$$[\varepsilon] = \{x \mid \#_a(x) \bmod 4 = 0\}$$

$$[a] = \{x \mid \#_a(x) \bmod 4 = 1\}$$

$$[a^2] = \{x \mid \#_a(x) \bmod 4 = 2\}$$

$$[a^3] = \{x \mid \#_a(x) \bmod 4 = 3\}$$

There is one equivalence class for each state in a DFA. For, any DFA there are finitely many states and hence finitely many equivalence classes one for each state.

Now we could see that the relation $\equiv_A$ induced by the DFA is

1) an equivalence relation
2) right congruence.

3) refines $L(A)$

4) is of finite index.

This kind of relation satisfying all the above conditions, is called a Myhill-Nerode relation.

<u>Brief note</u>:

consider a binary relation $\equiv_A$ on Σ^* induced by $A = (Q, s, \delta, F)$ on Σ, defined as

$$\forall x, y \in \Sigma^*, \; x \equiv_A y$$
$$\implies \hat{\delta}(s, x) = \hat{\delta}(s, y)$$

if this relation satisfies the below conditions then it is a Myhill-Nerode Relation. (MNR).

1) $\equiv_A$ is an equivalence relation.

2) $\equiv_A$ is a right congruence.

3) $\equiv_A$ refines $L(A)$.

4) $\equiv_A$ is of finite index.

7

another example:

$L = \{ x \in \{a\}^* \mid 3 \text{ divides } |x| \}$

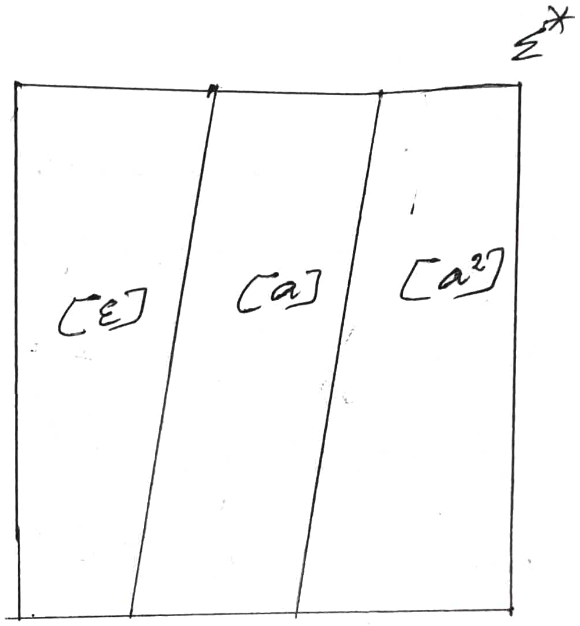
$x \equiv y \implies |x| \bmod 3 = |y| \bmod 3.$

Equivalence classes are

$[\varepsilon] = \{ a^i \mid i \% 3 = 0 \}$

$[a] = \{ a^i \mid i \% 3 = 1 \}$

$[a^2] = \{ a^i \mid i \% 3 = 2 \}$



MN Relation for Regular Language.

There exists an MN relation for every regular language L over an alphabet set Σ.

proof:

Suppose L is regular, we need to prove that there is a myhill-nerode relation. So when L is regular then there exists a DFA A such that $L = L(A)$

consider the binary relation $\equiv_A$ on Σ induced by $A = (Q, s, \delta, F)$ on Σ defined as

$$\forall x, y \in \Sigma^*, \ x \equiv_A y$$
$$\Longleftrightarrow \hat{\delta}(s, x) = \hat{\delta}(s, y)$$

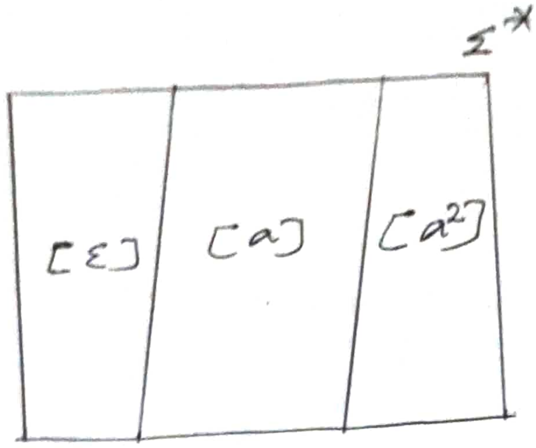
we proved that $\equiv_A$ is an MN Relation.

So we proved that whenever L is regular there is a DFA A, if there is a DFA A the relation induced by it $\equiv_A$ is a myhill Nerode relation. Hence we get MN Relation on set of strings from Σ^*.

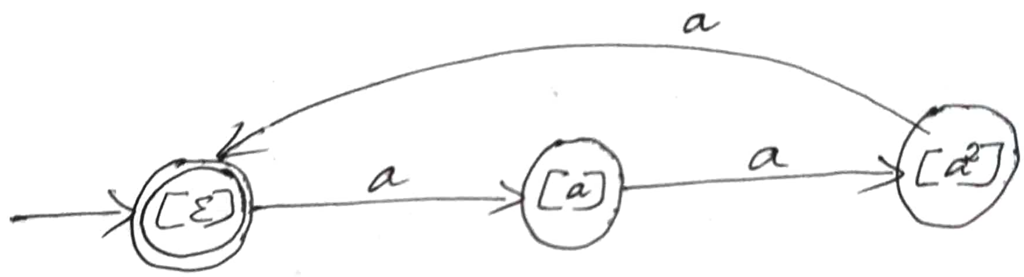
MN Relation to DFA.

Eg: consider the MN relation represented by the following equivalence classes for the Language $L = \{x \in \{a\}^* \mid 3 \text{ divides } |x|\}$

(9)



Each equivalence class of the myhill-Nerode relation is represented as a state in the DFA.



$$Q = \{ [x] \mid x \in \Sigma^* \}$$

$$s \mid q_0 = [\varepsilon]$$

$$\delta([x], a) = [xa]$$

$$F = \{ [x] \mid x \in L \}$$

Here the string is in the Language, if its length is divisible by 3. So the equivalence class $[\varepsilon]$.

So $[\varepsilon]$ (state corresponding to $[\varepsilon]$) becomes final state.

Formal representation (MNR to DFA)

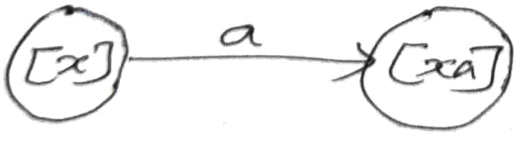
Given an MN relation $\equiv$ for a Language L over an alphabet set Σ one can automatically construct a DFA $A_{\equiv} = (Q, s, \delta, F)$, such that $L(A_{\equiv}) = L$.

$$Q = \{ [x] \mid x \in \Sigma^* \}$$

$$s = [\varepsilon]$$

$$\delta([x], a) = [xa]$$

$$F = \{ [x] \mid x \in L \}$$



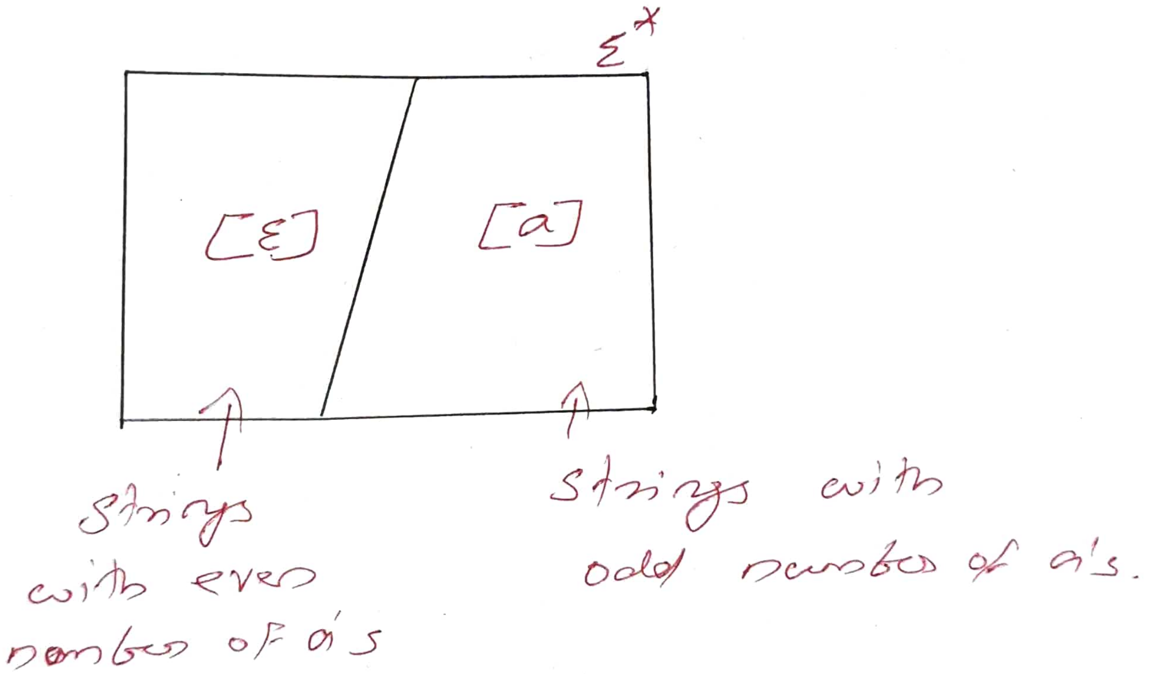
Eg 2: (MNR to DFA)

consider the MN relation defined as $x \equiv y$ if and only if $\#_a(x) \bmod 2 = \#_a(y) \bmod 2$

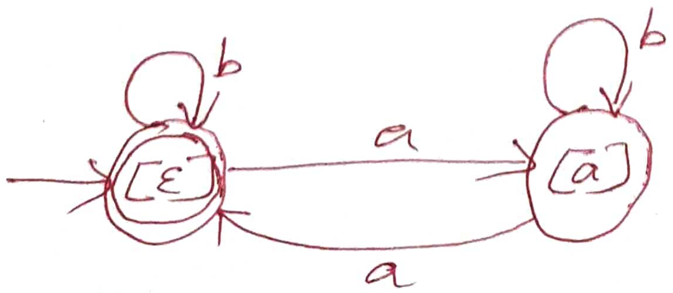
(11) on strings over $\Sigma = \{a, b\}$ and the language $L = \{x \in \{a,b\}^* \mid x$ contains even number of a's$\}$

There are 2 equivalence classes:

[no. of a's even & no. of a's odd]



* Each equivalence class becomes a state.
* The state represented by $[\varepsilon]$ becomes the initial state.

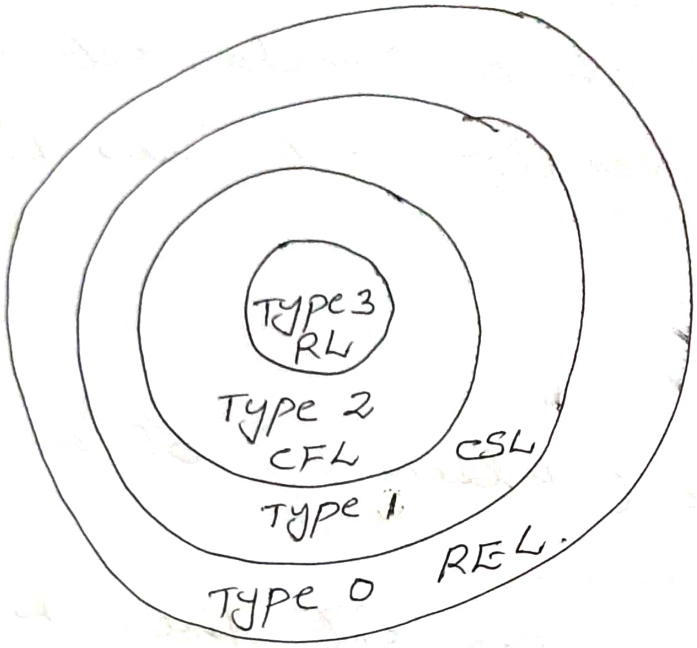


$[\varepsilon . a] = [a]$

$[\varepsilon . b] = [b] = [\varepsilon]$

$[a . b] = [ab] = \boxed{a}$

1 (7) Chomsky hierarchy:



Type 3 - Regular Grammar → Regular Language

Type 2 - context free Grammar → context free Language

Type 1 - context sensitive Grammar → context sensitive Language.

Type 0 - unrestricted Grammar → Recursively Enumerable languages.

Type 3 - FA

Type 2 - PDA

Type 1 - LBA

Type 0 - Turing machine.

Context Free Grammar (CFG)

A Grammar $G = (V, T, P, S)$ is said to be context free if all productions in P have the form $\alpha \rightarrow \beta$ where $|\alpha| \leq |\beta|$ and α is an element of V. (The LHS contains only one variable)

Eg1:

consider a Grammar $G = (V, T, P, S)$

where $V = \{S\}$

$T = \{a, b\}$

$P = \{S \rightarrow aSbS,\ S \rightarrow bSaS,\ S \rightarrow \varepsilon\}$

$S = S$

$V \rightarrow$ Finite set of variables (Non-terminals)

$T \rightarrow$ Finite set of Terminals

$P \rightarrow$ set of productions.

$S \rightarrow$ start symbol $\in V$

Applications:

1) For defining programming languages
2) For construction of compilers
(For parsing the program by constructing syntax tree).

CONTEXT FREE LANGUAGES (CFL)

The language Generated using CFG is called context free language.

properties:

1) context free languages are closed under union.

ie, if L_1 and L_2 are 2 context free languages then $L_1 \cup L_2$ is also a context free language (CFL).

2) context free languages are closed under concatenation.

ie, if L_1 and L_2 are 2 context free languages then $L_1 \cdot L_2$ is also a context free language.

3) context free languages are closed under kleen closure.

ie, if L_1 and L_2 are 2 context free languages then L_1^* and L_2^* are also context free languages.

4) context free languages are not closed under intersection and complement.

ie, if L_1 and L_2 are 2 CFL, then

$L_1 \cap L_2$ is not a CFL

L_1' and L_2' are not CFL

* The family of regular language is a proper subset of the family of context free language.

* Each context free language is accepted by a push down automata (PDA).

DERIVATION:

Derivation is a sequence of production rules. It is used to get the input string through the productions of the CFG.

* 2 things to be considered while the derivation is

1) Select a non-terminal which is to be replaced.

2) Select the production by which the non-terminal is to be replaced, (if there are multiple productions for the same Non-terminal).

There 2 types of derivation.

1) Leftmost derivation

2) Rightmost derivation.

Leftmost derivation is obtained by applying production to the leftmost variable (Non-terminal) in each step.

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

Eg: (Leftmost derivation)

Grammar:

$S \rightarrow AB \mid \epsilon$

$A \rightarrow aB$

$B \rightarrow Sb$

Derive the string "abb" using leftmost derivation.

$S \Rightarrow AB$
$\Rightarrow aBB$
$\Rightarrow aSbB$
$\Rightarrow abB$
$\Rightarrow abSb$
$\Rightarrow abb$

rightmost derivation

string : abb, same grammar.

$S \Rightarrow AB$
$\Rightarrow ASb$
$\Rightarrow Ab$
$\Rightarrow aBb$
$\Rightarrow aSbb$
$\Rightarrow abb$.

12 * **Derivation tree:**

Derivation tree is a graphical representation for the derivation of the given production rules for a given CFG.

The derivation tree is also called parse tree.

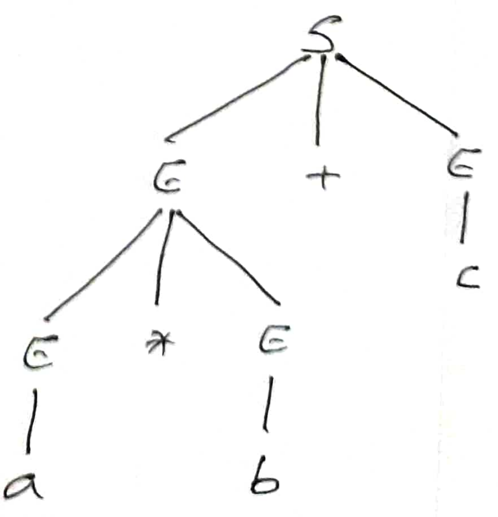
A parse tree has the following properties:

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

Eg:

$E \rightarrow E+E \mid E*E \mid a \mid b \mid c$

input : $a*b+c$



$E \Rightarrow E+E$

$\Rightarrow E*E+E$

$\Rightarrow a*E+E$

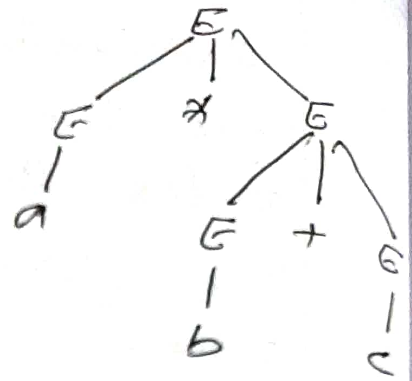
$\Rightarrow a*b+E$

$\Rightarrow a*b+c$

others way

$$E \Rightarrow E * E$$
$$\Rightarrow a * E$$
$$\Rightarrow a * E + E$$
$$\Rightarrow a * b + E$$
$$\Rightarrow a * b + c$$

parse tree:



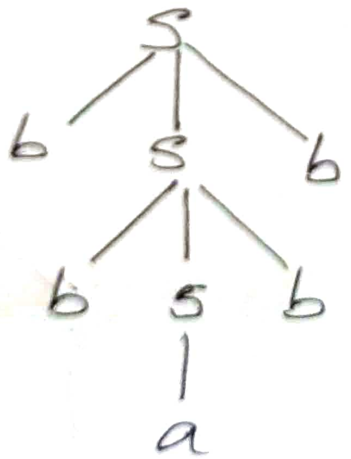
Eg2:

$S \rightarrow bSb \mid a \mid b$

draw a derivation tree for the string "bbabb"

Now start with our start symbol.

$$S \Rightarrow bSb \quad (S \rightarrow bSb)$$
$$\Rightarrow bbSbb \quad (S \rightarrow a)$$
$$\Rightarrow bbabb$$



Eg 3:

$S \rightarrow aB \mid bA$

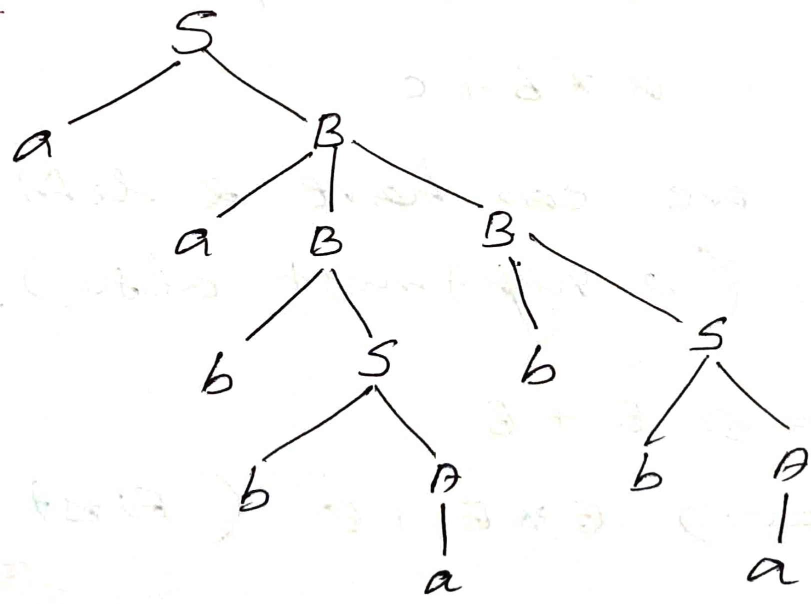
$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$

construct a derivation tree for the string "aabbabba".

$$\begin{aligned}
S &\Longrightarrow aB \\
&\Longrightarrow aaBB \\
&\Longrightarrow aabSB \\
&\Longrightarrow aabbAB \\
&\Longrightarrow aabbaB \\
&\Longrightarrow aabbabS \\
&\Longrightarrow aabbabbA \\
&\Longrightarrow aabbabba.
\end{aligned}$$

parse tree:



(15)

AMBIGUITY in GRAMMAR

* A Grammar is said to be ambigous if there exists.
  - → more than one leftmost derivation

    or
  - → more than one rightmost derivation

    or
  - → more than one parse tree for the given input string.

* if the grammar is not ambigous then it is called unambigous.

* if the grammar is ambigous, then it is not good for compiler construction.

Eg:

$E \rightarrow E+E \mid E*E \mid a \mid b \mid c$

input : $a*b+c$.

Here we can have 2 left most derivations (2 rightmost also).

1) $E \Longrightarrow E+E$

$\Longrightarrow E*E+E$ (First E is replaced by E*E)

$\Longrightarrow a*E+E$

$\Longrightarrow a*b+E$

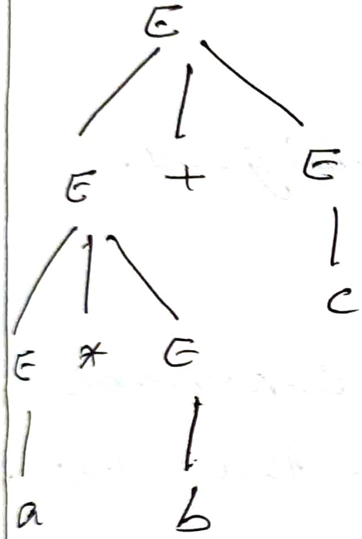
$\Longrightarrow a*b+c$

2)

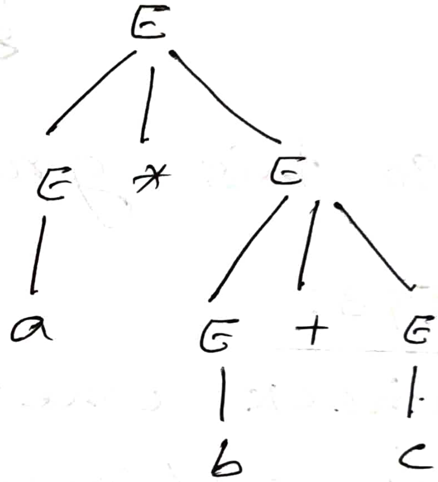
$$E \Rightarrow E * E$$
$$\Rightarrow a * E$$
$$\Rightarrow a * E + E$$
$$\Rightarrow a * b + E$$
$$\Rightarrow a * b + c.$$

using this we can construct 2 parse tree

1



2



so this grammar is ambigous.

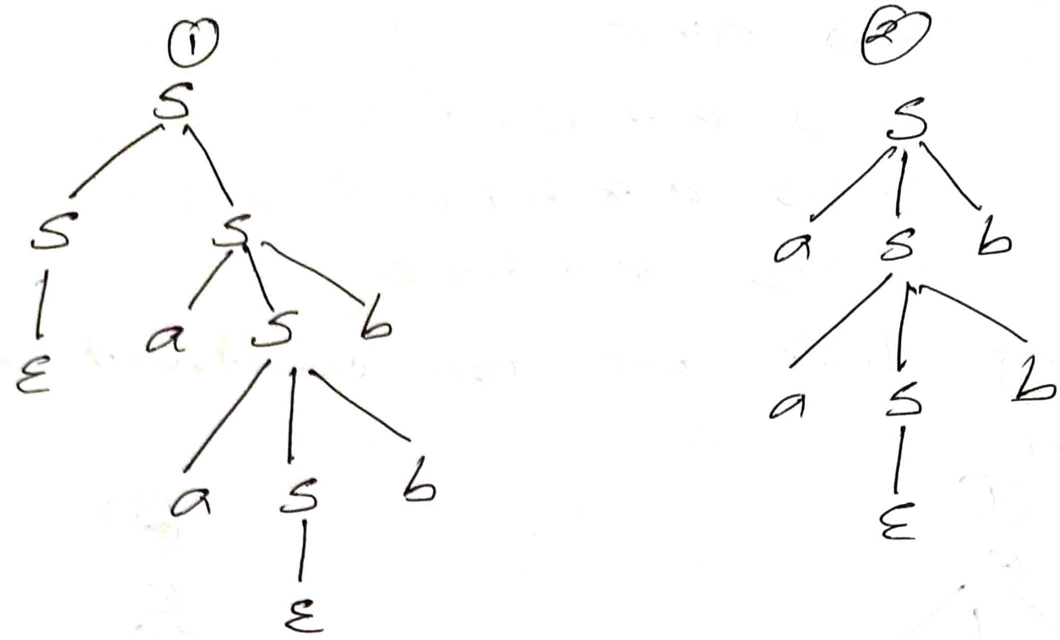
Eg 2:

check whether the given string is ambigous or not for the given string. "aabb".

$$S \rightarrow aSb \mid SS$$
$$S \rightarrow \varepsilon$$

Here also we can get 2 parse trees.

if you get 2 parse trees or 2 leftmost derivation or 2 rightmost derivation thus the grammar is ambigous.



so this grammar is ambigous.

Eg 3:

check whether the given Grammar G is ambigous or not for the input string. "a(a)aa"

A ⟶ AA

A ⟶ (A)

A ⟶ a

Here, check whether this string has 2 leftmost derivation.

(1) A ⟹ AA

⟹ AAA (First A ⟶ AA)

⟹ AAAA (First A ⟶ AA)

⟹ aAAA (First A ⟶ a)

⟹ a(A)AA (Second A ⟶ (A))

18

$\Rightarrow a(a)AA$ (second $A \to a$)

$\Rightarrow a(a)aA$ (third $A \to a$)

$\Rightarrow a(a)aa$ (last $A \to a$)

2

$A \Rightarrow AA$

$\Rightarrow aAA$

$\Rightarrow a(A)A$

$\Rightarrow a(a)A$

$\Rightarrow a(a)AA$

$\Rightarrow a(a)aA$

$\Rightarrow a(a)aa.$

So we got 2 leftmost derivation.
So this grammar is ambigous.

Sentential Form: Every string of symbols in the derivation is a sentential form.
A Sentence is a sential form that has only terminal symbols:

Eg: $E \to E+E \mid E*E \mid a \mid b \mid c$ i/p : $a+b*c$

$E \Rightarrow E+E$

$\Rightarrow E+E*E$

$\Rightarrow a+E*E$

$\Rightarrow a+b*E$

$\Rightarrow a+b*c$

Sentential Form

Sentence.

(19) if an input string w is derived from a start symbol S using multiple derivation steps then it is shown as

$$S \xRightarrow{*} w.$$

* SIMPLIFICATION OF CFG.

All the Grammar symbols are not always optimized, ie, the grammar may consist of some extra symbols. Having extra symbols increases the length of the grammar.

*) Simplification of grammar means reduction of grammar by removing useless symbols.

* The preliminary simplification which are applied on grammars to convert them to reduced grammar are

1) Elimination (removal) of useless symbols.

2) Elimination of ε- productions.

3) Elimination of unit productions.

Elimination of useless symbols:

A symbol is useless if it does not appear on the right-hand side of the production rule and does not take part in the derivation of any string.

consider the eg: $S \to abS \mid abA \mid abB$

$A \to cd$

$B \to aB$

$C \to dc$

in this example "$C \to dc$" is useless because the variable "C" will never occurs in derivation of any string.
similarly the production "$B \to aB$" is also useless because there is no way it will ever terminate. if it never terminates, then it can never produce a string. Hence the production can never take part in any derivation. To remove these productions, we first find all the variables which will never lead to a terminal string such as "B". we then remove all the productions in which "B" occurs.
Here we remove the productions of "C" as well.
So the resultant Grammar

$S \to abS \mid abA$

$A \to cd$

Elimination of ε-productions.

The productions of the type $S \to \varepsilon$ are called ε-production. These type of productions

21

can only be removed from those grammars that do not generate "ε".

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

Step 2: For each production $A \to \alpha$, construct all productions $A \to x$, where x is obtained from α by removing one or more non-terminal (variable) from step 1.

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

Eg:

$S \to ABC$
$A \to aA \mid \varepsilon$
$B \to bB \mid \varepsilon$
$C \to c$

First identify the nullable variables.

Here nullable variables are A, B.

now for each production, do the following.

$S \to ABC \mid BC \mid AC \mid C$
$A \to aA \mid a$
$B \to bB \mid b$
$C \to c$

Another eg:

$S \rightarrow aS \mid A$

$A \rightarrow \varepsilon$

Here the language contains "ε" (because "ε" can be generated from this grammar as its string).

So we can not remove "ε" from this.

Removing (eliminating) unit productions:

Any production of context-free grammar of the form

$A \rightarrow B$,

where $A, B \in V$ is called a unit-production.

To remove such productions, add $A \rightarrow x$ to the grammar rule wherever $B \rightarrow x$ occurs in the grammar.

Then delete $A \rightarrow B$ from the grammar.

Repeat ~~the~~ above procedure until all unit productions are removed.

Eg:

$S \rightarrow 0A \mid 1B \mid C$

$A \rightarrow 0S \mid 00$

$B \rightarrow 1 \mid A$

$C \rightarrow 01$

Here $S \rightarrow C$ is a unit production.

while removing $S \to C$ we have to consider what C gives. so we can add rule to S.

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

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

$$B \to 1 \mid 0S \mid 00$$

So our final Grammar

$$S \to 0A \mid 1B \mid 01$$
$$A \to 0S \mid 00$$
$$B \to 1 \mid 0S \mid 00$$
$$C \to 01$$

Here $C \to 01$ can be removed as C can not be reached from S.

$$S \to 0A \mid 1B \mid 01$$
$$A \to 0S \mid 00$$
$$B \to 1 \mid 0S \mid 00.$$

Examples:

1) Eliminate useless symbols and productions from $G = (V, T, P, S)$, where $V = \{S, A, B, C\}$ and $T = \{a, b\}$ with P consists of

$$S \to aS \mid A \mid C,$$
$$A \to a,$$
$$B \to aa$$
$$C \to aCb$$

(24) First identify the variables that can lead to a terminal string. Here $A \to a$, $B \to aa$ can lead to terminal string so A, B are these type of variables.
But C is not this type because from C we can never generate a terminal string. So remove C and its productions.
So we get

$$S \to aS \mid A$$
$$A \to a,$$
$$B \to aa$$

Now find the variables that can not be reached from the start variable.
Here "B" can not be reached from S.
So B is useless. So remove B and its productions.
So our resultant grammar

$$S \to aS \mid A$$
$$A \to a$$

2) Find a context free grammar without ε-productions equivalent to the grammar defined by.

$S \to ABaC$, $\quad B \to b \mid \varepsilon$, $\quad D \to d$.
$A \to BC$, $\quad C \to D \mid \varepsilon$,

(25) Here we have to identify the nullable variables. Here we have A, B, C as nullable variables. (26)

$S \to ABaC \mid BaC \mid Aa\epsilon \mid ABa \mid aC \mid Aa \mid Ba \mid a$.

$A \to BC \mid B \mid C$

$B \to b$.

$C \to D$.

$D \to d$.

3) Remove all unit-productions from

$S \to Aa \mid B$.

$B \to A \mid bb$

$A \to a \mid bc \mid B$

Here the unit productions are

$S \to B$

$B \to A$

$A \to B$

consider the productions of S

$S \to Aa \mid B$

Here we have to remove B

$S \to Aa \mid bb \mid A$

since A is also a unit production from B

Remove A too.

(26)

$S \to Aa | bb | a | bc$

similarly

$B \to A | bb.$

Here remove A by applying its productions other than unit production

$B \to bb | a | bc$

similarly

$A \to a | bc | B$

Here remove B.

$A \to a | bc | bb.$

So our resultant grammar

$S \to Aa | bb | bc | a$

$A \to a | bc | bb$

$B \to bb | a | bc.$

* Normal Forms

There are many kinds of normal forms we can establish for context free grammars.

Chomsky Normal Forms.

A context free grammar is in chomsky normal form if all production rules satisfy one of the following conditions

(27) * A non-terminal generating a terminal

Eg: $A \to a$

* A non-terminal generating 2 non-terminals

Eg: $A \to BC$

* start symbol generating ε.

$S \to \varepsilon$.

Eg: $S \to AS \mid a$

$A \to SA \mid b$.

is in chomsky normal form (CNF)

But the grammar $S \to AS \mid AAS$

$A \to SA \mid aa$

is not in CNF.

<u>Conversion of CFG to CNF</u>

Step 1: Eliminate start symbol from RHS.
if start symbol S is at the RHS of any production in the grammar, create a new production as:

$S' \to S$

where S' is the new start symbol

(28) Step 2: Eliminate null, unit or useless productions.

Step 3: Eliminate terminals from RHS if they exist with other terminals or non-terminals. Eg: production rule $X \rightarrow xY$ can be decomposed as

$$X \rightarrow ZY$$
$$Z \rightarrow x$$

Step 4: Eliminate RHS with more than two non-terminals.

Eg: production rule $X \rightarrow XYZ$ can be decomposed as

$$X \rightarrow PZ$$
$$P \rightarrow XY$$

Example:

$$S \rightarrow ASA \mid aB$$
$$A \rightarrow B \mid S$$
$$B \rightarrow b \mid \epsilon .$$

Step 1: Since S is appearing in RHS, we have to add a new production $S' \rightarrow S$, S' is the new start symbol.

(29) $S' \to S$

$S \to ASA \mid aB$

$A \to B \mid S$

$B \to b \mid \epsilon$.

2) remove the null productions.

Here the null productions are $B \to \epsilon$, since $B \to \epsilon$, & $A \to B$ we get another null production $A \to \epsilon$.

So we have to remove both the null productions $B \to \epsilon$, $A \to \epsilon$.

First remove $B \to \epsilon$

$S' \to S$

$S \to ASA \mid aB \mid a$

$A \to B \mid S \mid \epsilon$,

$B \to b$

Now remove $A \to \epsilon$

$S' \to S$

$S \to ASA \mid aB \mid a \mid SA \mid AS \mid S$

$A \to B \mid S$

$B \to b$

(30)

Now

3) Remove the unit productions.

The unit productions here are

$S' \to S$, $S \to S$, $A \to B$, $A \to S$.

First remove $S' \to S$

$S' \to ASA \mid aB \mid a \mid SA \mid AS \mid S$

$S \to ASA \mid aB \mid a \mid SA \mid AS \mid S$

Since $A \to B \mid S$

$B \to b$

now remove $S \to S$

So remove S and its productions. Since S is same on both sides we can just remove S.

$S' \to ASA \mid aB \mid a \mid SA \mid AS$

$S \to ASA \mid aB \mid a \mid SA \mid AS$

$A \to B \mid S$

$B \to b$

Now remove $A \to B$

$S' \to ASA \mid aB \mid a \mid SA \mid AS$

$S \to ASA \mid aB \mid a \mid SA \mid AS$

$A \to b \mid S$

$B \to b$

(31) Now remove $A \to S$.

$S' \to ASA \mid aB \mid a \mid AS \mid SA$,

$S \to ASA \mid aB \mid a \mid AS \mid SA$,

$A \to b \mid ASA \mid aB \mid a \mid AS \mid SA$,

$B \to b$.

4) now find the productions that has more than 2 variables in RHS.

$S' \to ASA$, $S \to ASA$, $A \to ASA$.

[now SA is given in the production as a new variable X.

$X \to SA$]

So our grammar is

$S' \to AX \mid aB \mid a \mid AS \mid SA$

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

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

$B \to b$.

$X \to SA$

5) now change the productions

$S' \to aB$, $S \to aB$, and $A \to aB$

to get that add a new production

$Y \to a$

(32) So our resultant grammar is

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

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

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

$B \rightarrow b$

$X \rightarrow SA$

$Y \rightarrow a$

Eg 2:

convert the grammar with productions

$S \rightarrow ABa,$

$A \rightarrow aab,$

$B \rightarrow Ac.$

to chomsky normal form (CNF)

Here we do not have any ε-productions or unit productions.

we introduce new variables X, Y, Z

$S \rightarrow ABX$

$A \rightarrow XXY$

$B \rightarrow AZ$

$X \rightarrow a$

$Y \rightarrow b$

$Z \rightarrow c$

in the second step we introduce additional variables D and E to get the first two productions into the normal form.

$S \rightarrow AD$

$D \rightarrow BX$

$A \rightarrow XE$

$E \rightarrow XY$

$B \rightarrow AZ$

$X \rightarrow a$

$Y \rightarrow b$

$Z \rightarrow c$

This is our final Grammar (IN CNF)

* GREIBACH NORMAL FORM (GNF)

A context free grammar is in Greibach normal form if all production rules satisfy one of the following conditions.

*) A non-terminal generating a terminal

eg: $A \rightarrow a$

*) A non-terminal generating a terminal followed by any number of non-terminals.

eg: $A \rightarrow aABCD\ldots$

*) Start symbol generating ε.

eg: $S \rightarrow \varepsilon$.

34

Conversion of CFG to GNF

Step 1: Convert the given grammar into CNF if the grammar is not in CNF.

Step 2: Eliminate left recursion from grammar if it exists.

Step 3: Convert the production rules into GNF form, if it is not in GNF, by proper substitution.

Eg: 1

$S \rightarrow AB,$

$A \rightarrow aA \mid bB \mid b,$

$B \rightarrow b.$

This grammar is not in GNF. we can convert this into GNF using substitution.

$S \rightarrow aAB \mid bBB \mid bB$

$A \rightarrow aA \mid bB \mid b$

$B \rightarrow b.$

Eg: 2.

$S \rightarrow abSb \mid aa,$

This grammar is not in GNF. Here we can introduce new variables A & B that are synonyms for a and b.

(35)

$S \rightarrow aBSB \mid aA$

$A \rightarrow a,$

$B \rightarrow b.$

Eg 3:

$S \rightarrow XB \mid AA$

$A \rightarrow a \mid SA$

$B \rightarrow b$

$X \rightarrow a$

This Grammar is not in GNF.
Here the productions $S \rightarrow XB$, $S \rightarrow AA$, and $A \rightarrow SA$ are not in GNF, but the remaining productions are in GNF.
Here we substitute $S \rightarrow XB \mid AA$ in $A \rightarrow SA$.

$S \rightarrow XB \mid AA$

$A \rightarrow a \mid XBA \mid AAA$

$B \rightarrow b$

$X \rightarrow a$

Here the production rules $S \rightarrow XB$, $S \rightarrow AAA$, $B \rightarrow XBA$, $B \rightarrow AAA$ violate the rules for a Grammar to be in GNF.

(36) But there is a production $X \to a$ that can be used to substitute the value of X in $S \to XB$, $A \to XBA$.

So our grammar becomes

$$S \to aB \mid AA$$
$$A \to a \mid aBA \mid AAA$$
$$B \to b$$
$$X \to a$$

now there are 2 productions $S \to AA$, and $A \to AAA$ which are not in CNF. So first eliminate the left recursion from $A \to AAA$.

The rule to eliminate left recursion is

$$\boxed{\begin{array}{l} A \to A\alpha \mid \beta \\ \quad\Downarrow \\ A \to \beta A' \\ A' \to \alpha A' \mid \varepsilon \end{array}}$$

So here

$$A \to aA' \mid aBAA'$$
$$A' \to AAA' \mid \varepsilon$$

So our grammar will be

$S \to aB \mid AA$ $\qquad$ $A' \to AAA' \mid \varepsilon$

$A \to aA' \mid aBAA'$, $\qquad$ $B \to b$

(37) now we got an ε- production ($A' \rightarrow \varepsilon$).
So we have to eliminate "ε" production

$S \rightarrow aB \mid AA$

$A \rightarrow aA' \mid aBAA' \mid a \mid aBA$

$A' \rightarrow AAA' \mid AA$

$B \rightarrow b$

$X \rightarrow a$

The production rule $S \rightarrow AA$ is not in GNF, so we substitute

$A \rightarrow aA' \mid aBAA' \mid a \mid aBA$ in

$S \rightarrow AA$

So our grammar becomes.

$S \rightarrow aB \mid aA'A \mid aBAA'A \mid aA \mid aBAA$

$A \rightarrow aA' \mid aBAA' \mid a \mid aBA$

Similarly substitute

$A \rightarrow aA' \mid aBAA' \mid a \mid aBA$ in

$A' \rightarrow AA$ also

$A' \rightarrow AAA' \mid aA'A \mid aBAA'A \mid aA \mid aBAA'$

$B \rightarrow b$

$X \rightarrow a$

(38)

So our grammar becomes.

$S \rightarrow aB \mid aA'A \mid aBAA'A \mid aA \mid aBAA$

$A \rightarrow aA' \mid aBAA' \mid a \mid aBA$

$A' \rightarrow AAA' \mid aA'A \mid aBAA'A \mid aA \mid aBAA$

$B \rightarrow b$

$X \rightarrow a$

The production rule $A' \rightarrow AAA'$ is not in CNF, so we substitute

$A \rightarrow aA' \mid aBAA' \mid a \mid aBA$ in production

rule $A' \rightarrow AAA'$

$S \rightarrow aB \mid aA'A \mid aBAA'A \mid aA \mid aBAA$

$A \rightarrow aA' \mid aBAA' \mid a \mid aBA$

$A' \rightarrow aA'AA' \mid aBAA'AA' \mid aAA' \mid aBAAA'$

$\mid aA'A \mid aBAA'A \mid aA \mid aBAA$

$B \rightarrow b$

$X \rightarrow a$

So this grammar is in CNF.

MODULE IV

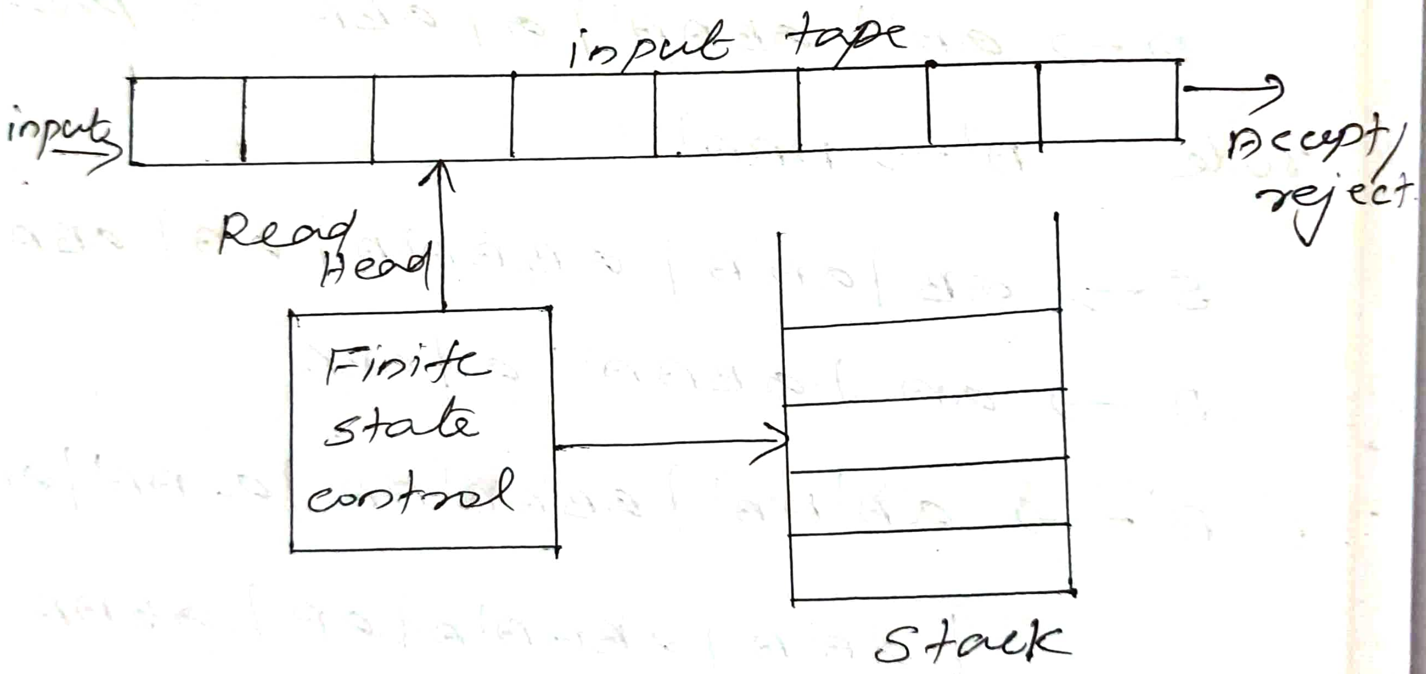
1

MODULE - IV

* PUSHDOWN AUTOMATA.

pushdown automata is a Finite automata with extra memory called stack which helps pushdown automata to recognize context free languages (CFL)

Representation of PDA



The pushdown automata consists of the following:

1. State control unit (Finite state control)
2. Reading Head.
3. Input tape.
4. Stack.

(2) The finite state control represents the states and the transition function, while the read head is used to read the input string starting from the left and towards right.

The input tape contains the input string.

Stack is a structure in memory in which we can push and pop the items from one end only.

In PDA, stack is used to store items temporarily.

Formal definition of PDA

A PDA can be represented by 7-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

$Q \rightarrow$ set of states.

$\Sigma \rightarrow$ is set of input symbols.

$\Gamma \rightarrow$ finite set of stack alphabet (set of symbols which can be pushed to / popped from stack)

$q_0 \longrightarrow$ initial state

$z \longrightarrow$ stack start symbol ($z \in \Gamma$)

$F \longrightarrow$ finite set of final states.

$\delta \longrightarrow$ Transition function which maps

3

$Q \times (\Sigma \cup \epsilon) \times \Gamma$ into $Q \times \Gamma^*$. In a given state, PDA will read input symbol and stack symbol and move to a new state and change the symbol of stack.

The transition function of PDA depends on the current state, input symbol and the symbol on the top of the stack. The transition function δ take triple argument and output of δ is a finite set of pair of arguments ie,

$$\delta(q, a, x) = (p, \gamma)$$

$q \rightarrow$ state in Q.

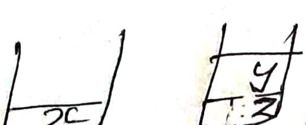
$a \rightarrow$ input symbol (ϵ is also included)

$x \rightarrow$ stack symbol at the top, $x \in \Gamma$.

$p \rightarrow$ new state after reading 'a'

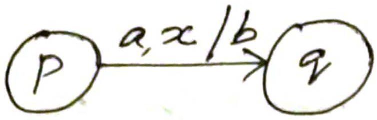
$\gamma \rightarrow$ symbol that is pushed onto stack that replaces the top symbol "x" on the stack.

if $\gamma = \epsilon$ then the stack is popped. if $\gamma = x$ then the stack is unchanged. if $\gamma = yz$ then x is replaced by z and y is pushed on to the stack



Graphical representation of PDA

Eg: $\delta(p, a, x) = (q, b)$



An arrow labelled start indicate starting state and double circled states are final states.

Eg: draw the transition diagram for the following

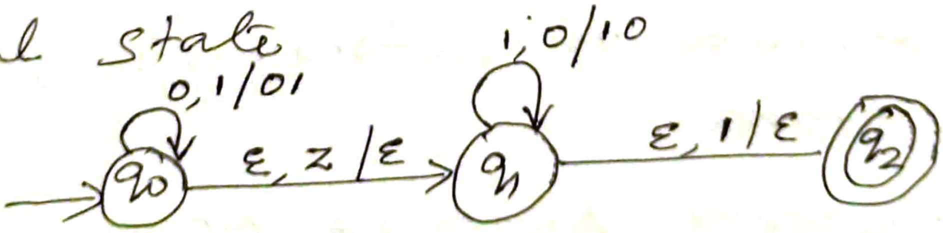
$\delta(q_0, 0, 1) = (q_0, 01)$

$\delta(q_0, \varepsilon, z) = (q_1, \varepsilon)$

$\delta(q_1, 1, 0) = (q_1, 10)$

$\delta(q_1, \varepsilon, 1) = (q_2, z)$

where q_0 is initial state and q_2 is final state



* **Non-Deterministic pushdown automata (NPDA) & Deterministic pushdown automata (DPDA)**

(5) Both NPDA and DPDA has the same tuple representation (7-tuple).
The main difference between NPDA & DPDA are

1) In DPDA there should be only one move from a state on an input symbol and stack symbol.
The NPDA can have more than one move from a state on input symbol and stack symbol.

2) The second difference is in DPDA when an ε-move is possible for some states, then no input consuming alternative should be there. But in NPDA that is allowed.

3) It is not always possible to convert non-deterministic pushdown automata to deterministic pushdown automata.

4) Expressive power of NPDA is more compared to expressive power of DPDA.

(6) Generally by definition PDA's are non-deterministic in nature. In NPDA we can have

$$\delta(p, a, x) = \{ (q_1, r) (q_2, r \cdot\cdot \}$$

That means multiple transitions allowed. But a PDA is said to be deterministic then

1) $\delta(p, a, x)$ contains at most one element / member ie,

$$\delta(p, a, x) = (q, r)$$

2) if $\delta(p, a, x)$ is non-empty, for some a in Σ, then $\delta(p, \varepsilon, x)$ must be empty.

Instantaneous description of PDA

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

ID is triple (q, w, α) where

q is the current state.

w is the remaining input (unconsumed input)

α is stack contents, top at the left.

Turnstile notation.

$\vdash$ sign is called turnstile notation and represents one move.

$\vdash^*$ represents a sequence of moves.

Eg: $(p, b, T) \vdash (q, w, \alpha)$

This implies that while taking a transition from a state p to state q, the input symbol 'b' is consumed, and the top of the stack 'T' is replaced by a new string 'α'.

Eg 1:

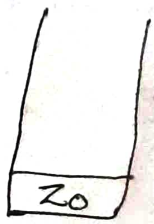
Design a PDA for the language

$$L = \{a^n b^n \mid n \geq 1\}$$

Here the number of a's and b's are same.

eg: i/p | a | a | a | b | b | b | ε |

initially



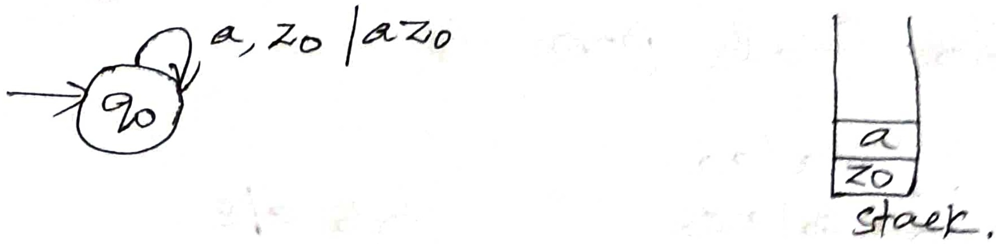
stack

Here the idea is to push whenever we read an "a" and whenever a "b" comes we need to

⑧ perform "pop" operation.

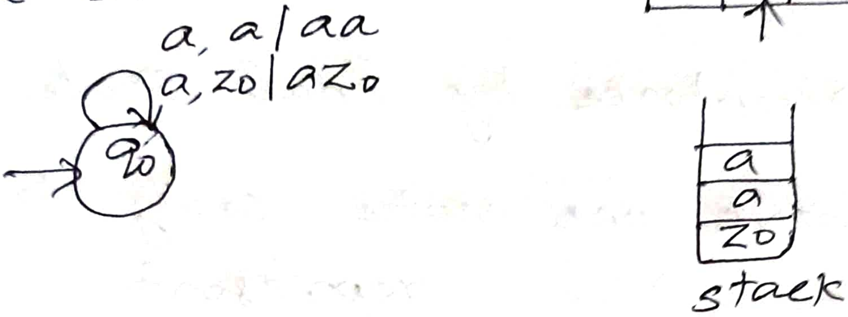
| a | a | a | b | b | b | ε |

Initially our PDA is in initial state, on first "a"



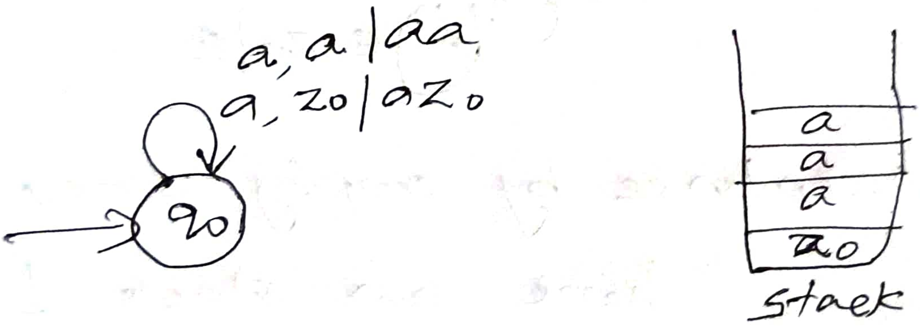
In the second "a"

| $\cancel{a}$ | a | a | b | b | b | ε |

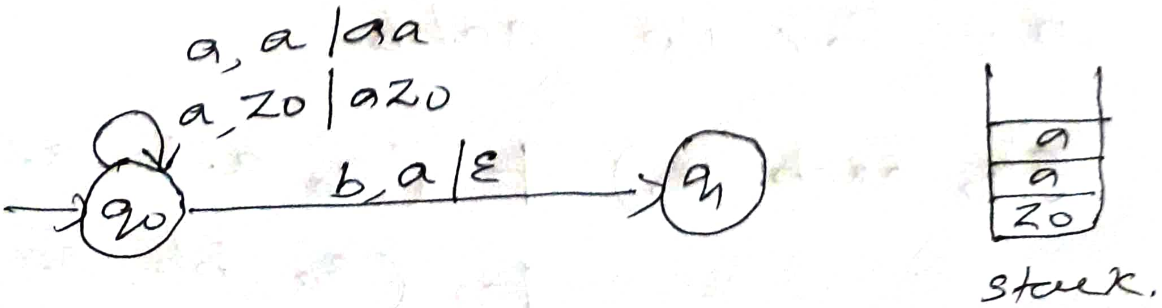


on third "a"

| $\cancel{a}$ | $\cancel{a}$ | a | b | b | b | ε |

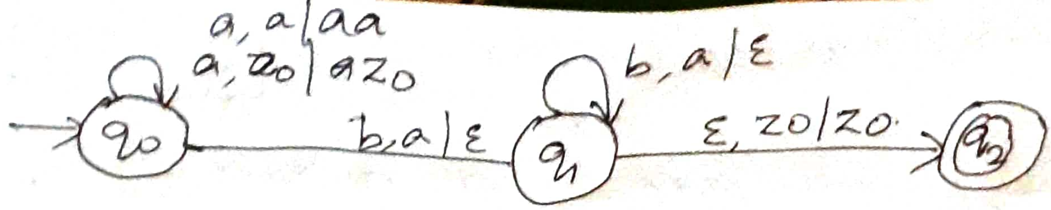


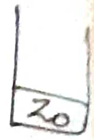
Then when we read our first "b" we need to perform "pop" operation



when the next b comes (second & third)

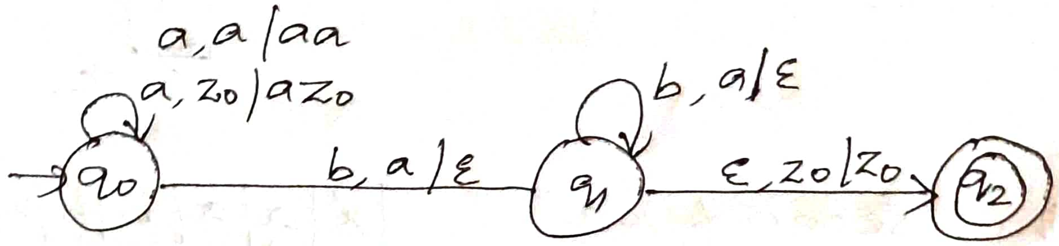
(a)





So our final PDA is

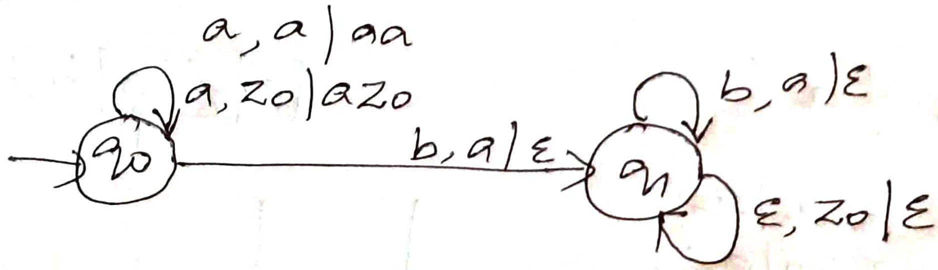
①



This is acceptance by final state.

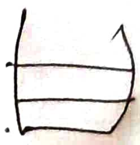
we can have one more way to represent

②



This is acceptance by empty stack

here our stack becomes empty.



Instantaneous description

ID: For i/p aabb.

$(q_0, aabb, Z_0) \vdash (q_0, abb, aZ_0)$

$\vdash (q_0, bb, aaZ_0)$

$\vdash (q_1, b, aZ_0)$

10

$\vdash (q_1, \varepsilon, Z_0)$

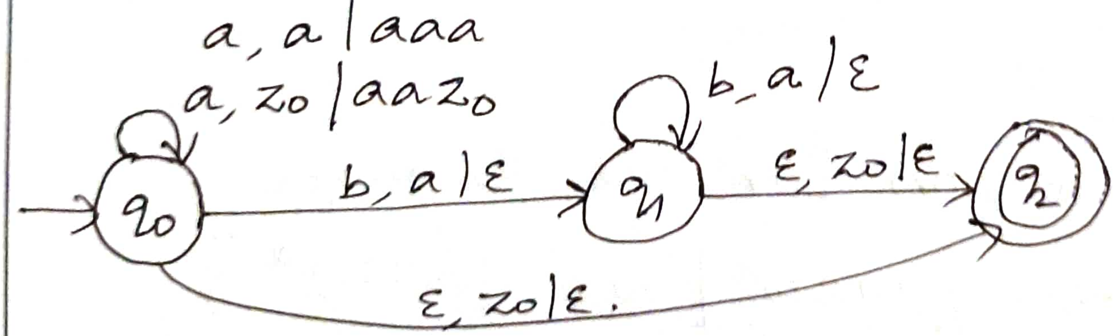
$\vdash (q_2, Z_0)$ or (q_1, ε).

Eg 2:

Design a PDA to accept the language $L = \{a^n b^{2n} \mid n \geq 0\}$ accepting by final state.

$L = \{\varepsilon, abb, aabbbb, \ldots\}$

Here the logic for PDA is when we read a single "a" we push (insert) two "a's" to the stack.
When we read a "b" we pop "a" from the top of the stack and when reading Z_0 on the stack we reach final state.



Transition Function:
$$\delta(q_0, \varepsilon, Z_0) = (q_2, \varepsilon)$$
$$\delta(q_0, a, Z_0) = (q_0, aaZ_0)$$
$$\delta(q_0, a, a) = (q_0, aaa)$$
$$\delta(q_0, b, a) = (q_1, \varepsilon)$$
$$\delta(q_1, b, a) = (q_1, \varepsilon)$$
$$\delta(q_1, \varepsilon, Z_0) = (q_2, \varepsilon)$$

(11)

Let us check the string "aabbbb" using ID.

$$(q_0, aabbbb, Z_0) \vdash (q_0, abbbb, aaZ_0)$$
$$\vdash (q_0, bbbb, aaaaZ_0)$$
$$\vdash (q_1, bbb, aaaZ_0)$$
$$\vdash (q_1, bb, aaZ_0)$$
$$\vdash (q_1, b, aZ_0)$$
$$\vdash (q_1, \varepsilon, Z_0)$$
$$\vdash (q_2, \varepsilon, \varepsilon)$$

So we reached our final state q_2. Hence the string is accepted.

Eg3:

coin question ✗.

Design a PDA for the language

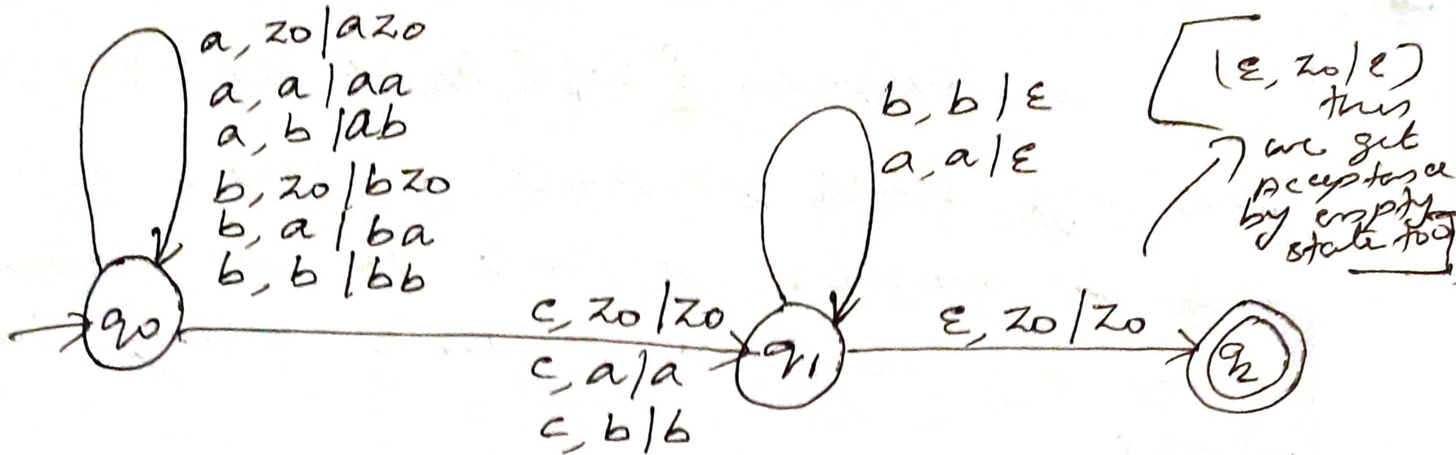
$$L = \{ wcw^R \mid w \in (a+b)^* \}$$

Eg: $w = abb$
$w^R = bba$

our string : abbcbba

$w = ba$
$w^R = ab$

our string : bacab.



Acceptance by final state

Language Acceptance by A PDA

The 2 methods of accepting a string in pushdown automata (PDA) are as follows.

1. Acceptance by final state.
2. Acceptance by empty stack.

Acceptance by final state.

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ be a PDA. Then $L(P)$, the language accepted by P by final state is

$$L(P) = \{ w \mid (q_0, w, z_0) \vdash^* (q, \varepsilon, \alpha) \}$$

for some state q in F and any stack string α.

Acceptance by empty stack.

For each PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

(13) we also define

$L = N(P) = \{ w \mid (q_0, w, Z_0) \vdash (q, \varepsilon, \varepsilon) \}$

for any state q. That is, N(P) is the set of inputs w that P can consume and at the same time empty its stack.

* Equivalence of PDA acceptance by empty stack & final state

Here we have to prove that if there is a PDA M_1 which accepts string by final state and M_2 is a PDA which accepts string by empty stack thus $L(M_1) = L(M_2)$.

Let $M_1 = (Q, \Sigma, \delta_1, q_0, \Gamma, Z_0, F)$

$M_2 = (Q \cup \{q_0', q_e\}, \Sigma, \delta_2, q_0', X_0, \Gamma \cup X_0, \phi)$

Case 1:

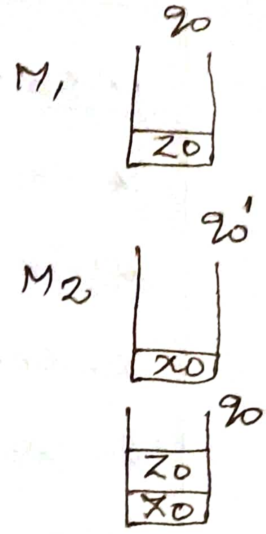
Here we have to prove that if M_1 accepts a particular string then M_2 will also that string.

Here our δ_1 is normal transition function of a PDA.

(14) but δ_2 is slightly different.

δ_2:

1) $\delta_2(q_0', \varepsilon, x_0) = (q_0, z_0 x_0)$



2) $\delta_2(q, a, z) = \delta_1(q, a, z)$

3) if M_1 reaches final state then M_2 will go to q_e

$\delta_2(q, \varepsilon, z) = (q_e, \varepsilon)$, $q \in F$ for M_1.

4) $\delta_2(q_e, \varepsilon, z) = (q_e, \varepsilon)$

part 2:

if M_2 accepts a particular string then M_1 will also accept the same string.

$M_1 = (Q \cup \{q_0', q_f\}, \Sigma, q_0', \delta_1, x_0, \Gamma \cup x_0, \underbrace{F}_{\{q_f\}})$

$M_2 = (Q, \Sigma, \delta_2, q_0', z_0, \Gamma, \phi)$

1) $\delta_1(q_0', \varepsilon, x_0) = (q_0, z_0 x_0)$

2) $\delta_1(q, a, z) = \delta_2(q, a, z)$.

3) $\delta_1(q, \varepsilon, x_0) = (q_f, \varepsilon)$ or (q_f, x_0)

[whenever M_2 accepts a string M_1 can also accept it].

Hence proved.

(15)

* EQUIVALENCE OF PDA & CFG

Theorem:

L is a context free language (CFL) if and only if there exists a PDA P such that L = L(P). and if L(P) is a language accepted by a PDA P then L(P) is context free.

part 1:

if L is context free, then some PDA recognizes it

proof ideas:

1. Let A be a CFL. By definition, A is generated by a CFG G.
2. we will show how to convert G into PDA P that accepts a string w if G generates w.
3. P will work by determining a derivation of w.

Here we use the stack of the PDA, to store the intermediate string generated over $(\Sigma \cup V)^*$

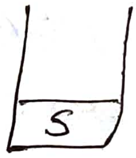
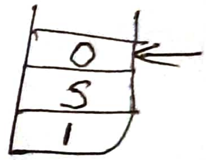
Start by pushing S onto the stack

* The leftmost symbol of the string is at the top of the stack
* we match off terminals by input symbols. and we replace a variable non-deterministically with one of its production rule.

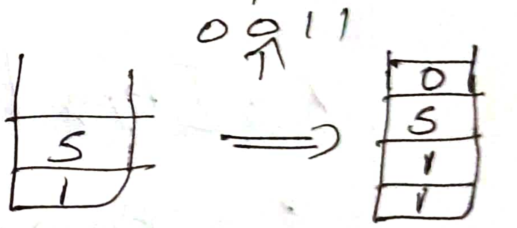
Eg:

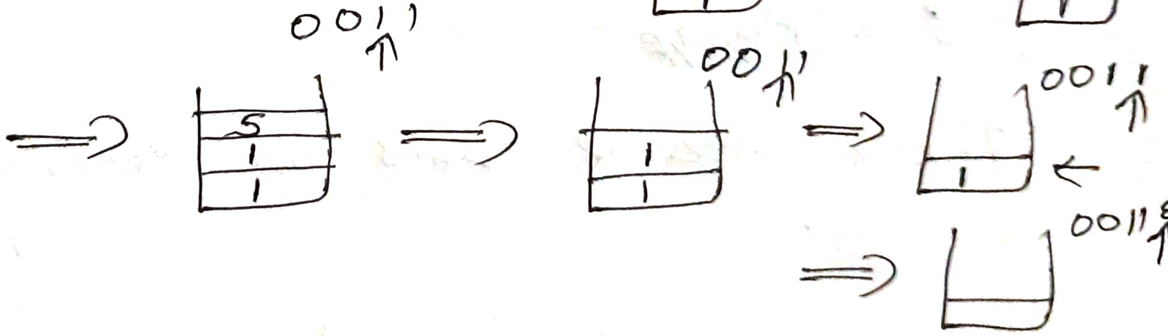
$S \to 0S1 \mid \varepsilon$

input:
0011

initially on stack  then 

then we match off "0" in the input and the stack top





part 2:

if a pushdown automata P recognizes a language then it is context free.

proof ideas:

we assume that P,
1) has a single accepting state q_f.

(17)

2) empties its stack before accepting

3) Each transition either pushes a symbol onto the stack or pops one off the stack, but does not do both simultaneously.

we build a corresponding Grammar G,
for each states p, q

variables are

A_{pq} = { strings that take the PDA from state p with an empty stack to state q on empty stack }

$V = \{ A_{pq} \mid p, q \in Q \}$

$S = A_{q_0 q_f}$

There are 2 cases of derivation from A_{pq}.

$A_{pq} \rightarrow a A_{rs} b$ for $r, s \in Q$.

$$\delta(p, a, \varepsilon) = (r, S)$$
$$\delta(s, b, S) = (q, \varepsilon)$$

we complete the grammar

$$A_{pp} \rightarrow \varepsilon$$

if A_{pq} generates x then x can bring

(8) p from (p, ε) to (q, ε) by reading x.
we can prove this by induction on the number of steps in derivation. From x to A_{pq}.

Similarly if p goes from (p, ε) to (q, ε) by reading x, thus A_{pq} generates x. This can be proved by induction on the number of steps in the computation of p that goes from (p, ε) to (q, ε) with x..

Converting CFG to PDA

we can construct a push down Automata that simulates the left most derivation of a CFG G.

$$G = (V, T, P, S)$$

to construct a PDA P from G that is Accept $L(G)$ by empty stack is as follows:

$$P = (\{q\}, T, V \cup T, \delta, q, S)$$

$Q = \{q\}$ $\qquad q$

$\Sigma = T$ $\qquad Z_0 = S$

$\Gamma = V \cup T$ $\qquad F = \phi$

δ

(19)

There are 2 rules.

1. For each non-terminal A in CFG

$\delta(q, \varepsilon, A) = \{(q, B)\}$ if $A \rightarrow B$ is a production of CFG.

2. For each terminal 'a' in CFG

$\delta(q, a, a) = \{(q, \varepsilon)\}$ if $a \in T$ in CFG.

Eg:

Find a PDA for the given grammar

$S \rightarrow 0S1 \mid 00 \mid 11$.

$Q = \{q\}$ $\quad \Gamma = \{0, 1, S\}$

$T = \{0, 1\}$ $\quad \delta, \; q_0, \; S, \; \phi$.

$q_0 \hookrightarrow q$.

δ is given by.

$\delta(q, \varepsilon, S) = \{(q, 0S1), (q, 00), (q, 11)\}$

$\delta(q, 0, 0) = \{(q, \varepsilon)\}$

$\delta(q, 1, 1) = \{(q, \varepsilon)\}$

The PDA will have only one state

Convert the given Grammar to PDA that accept the same language by empty stack.

$S \rightarrow 0S1 \mid A$ $\quad V = (A, S)$

$A \rightarrow 1A0 \mid S \mid \varepsilon$ $\quad T = \{0, 1, \varepsilon\}$

$P = \{Q, \Sigma, \delta, q_0, \Gamma, z_0, F)$

$F \rightarrow \phi$ (empty stack)

$Q = \{q_0\}$, $\Sigma = T = \{0, 1, \varepsilon\}$

$\Gamma = \{0, 1, \varepsilon, S, A\}$

δ, $q_0 = q$, $z_0 = S$, $F = \phi$

δ is given by

$\delta(q, \varepsilon, S) = \{(q, 0S1), (q, A)\}$

$\delta(q, \varepsilon, A) = \{(q, 1A0), (q, S), (q, \varepsilon)\}$

$\delta(q, 0, 0) = \{(q, \varepsilon)\}$

$\delta(q, 1, 1) = \{(q, \varepsilon)\}$

$\delta(q, \varepsilon, \varepsilon) = \{(q, \varepsilon)\}$.

Let us check for the string "0101"

$(q, 0101, S) \vdash (q, 0101, 0S1)$

$\vdash (q, 101, S1)$

21

$\vdash (q, 101, A1)$

$\vdash (q, 101, 1A01)$

$\vdash (q, 01, A01)$

$\vdash (q, 01, \varepsilon 01)$

$\vdash (q, 01, 01)$

$\vdash (q, 1, 1)$

$\vdash (q, \varepsilon, \varepsilon)$

Thus the string is accepted by the PDA by empty stack

Converting PDA to CFG

The production in P are induced by move of PDA as follows.

1) S productions are given by

$S \rightarrow [q_0 \, z_0 \, q]$ for $q \in Q$.

2) For every popping move

$\delta(q, a, z) = (q', \varepsilon)$

the corresponding production

$[q \, z \, q'] \rightarrow a$

3) For each push move

$$\delta(q, a, z) = (q_1, z_1 z_2 \cdots z_m)$$

the productions

$$[q\ z\ q'] = a\,[q_1\ z_1\ q_2]\,[q_2\ z_2\ q_3]\,[q_3\ z_3\ q_4] \cdots [q_m\ z_m\ q']$$

same

where each state $q', q_1, q_2 \cdots$ can be any state in Q.

Eg:

Generate a CFG for a given PDA M defined as $M = \{\{q_0, q_1\}, \{0, 1\}, \{x, z_0\}, \delta, q_0, z_0, q_1\}$ where

δ is given as follows.

1. $\delta(q_0, 1, z_0) = (q_0, xz_0)$
2. $\delta(q_0, 1, x) = (q_0, xx)$
3. $\delta(q_0, 0, x) = (q_0, x)$
4. $\delta(q_0, \varepsilon, x) = (q_1, \varepsilon)$
5. $\delta(q_1, \varepsilon, x) = (q_1, \varepsilon)$
6. $\delta(q_1, 0, x) = (q_1, xx)$
7. $\delta(q_1, 0, z_0) = (q_1, \varepsilon)$

1) S productions (rule 1)

$$S \rightarrow \underbrace{[q_0 Z_0 q_0]}_{A} \;/\; \underbrace{[q_0 Z_0 q_1]}_{B}$$

2) consider each transition

1) $\delta(q_0, 1, Z_0) = (q_0, xZ_0)$

This is a pushing move.

So the productions are

$$[q_0 Z_0 q_0] \rightarrow 1\, \overbrace{[q_0\ x\ q_0]}^{C}\ [q_0\ Z_0\ q_0]$$

$$[q_0\ Z_0\ q_0] \rightarrow 1\, \overbrace{[q_0\ x\ q_1]}^{D}\ \overbrace{[q_1\ Z_0\ q_0]}^{E}$$

$$[q_0\ Z_0\ q_1] \rightarrow 1\, [q_0\ x\ q_0]\ [q_0\ Z_0\ q_1]$$

$$[q_0\ Z_0\ q_1] \rightarrow 1\, [q_0\ x\ q_1]\ \underbrace{[q_1\ Z_0\ q_1]}_{H}$$

2) $\delta(q_0, 1, x) \rightarrow (q_0, xx)$

$$[q_0\ x\ q_0] \rightarrow 1\, [q_0\ x\ q_0]\ [q_0\ x\ q_0]$$

$$[q_0\ x\ q_0] \rightarrow 1\, [q_0\ x\ q_1]\ \overbrace{[q_1\ x\ q_0]}^{F}$$

$$[q_0\ x\ q_1] \rightarrow 1\, [q_0\ x\ q_0]\ \underbrace{[q_0\ x\ q_1]}_{G}$$

$$[q_0\ x\ q_1] \rightarrow 1\, [q_0\ x\ q_1]\ [q_1\ x\ q_1]$$

(24)

For 3) $\delta(q_0, 0, x) = (q_0, x)$

$$[q_0\ x\ q_0] \to 0\ [q_0\ x\ q_0]$$

$$[q_0\ x\ q_1] \to 0\ [q_0\ x\ q_1]$$

For 4) $\delta(q_0, \varepsilon, x) = (q_1, \varepsilon)$

This is a popping move

$$[q_0\ x\ q_1] \to \varepsilon$$

For 5) $\delta(q_1, \varepsilon, x) = (q_1, \varepsilon)$

$$[q_1\ x\ q_1] \to \varepsilon$$

For 6)

$\delta(q_1, 0, x) = (q_1, xx)$

productions.

$$[q_1\ x\ q_0] \to 0\ [q_1\ x\ q_0]\ [q_0\ x\ q_0]$$

$$[q_1\ x\ q_0] \to 0\ [q_1\ x\ q_1]\ [q_1\ x\ q_0]$$

$$[q_1\ x\ q_1] \to 0\ [q_1\ x\ q_0]\ [q_0\ x\ q_1]$$

$$[q_1\ x\ q_1] \to 0\ [q_1\ x\ q_1]\ [q_1\ x\ q_1]$$

For 7) $\delta(q_1, 0, Z_0) = (q_1, \varepsilon)$

$$A \leftarrow [q_1\ Z_0\ q_1] \to 0$$

(25)

we can rewrite the production using name.

$S \rightarrow A \mid B$

$A \rightarrow 1CA \mid 1DE$

$B \rightarrow 1CB \mid 1DH$

$C \rightarrow 1CC \mid 1DF \mid 0C \mid 0D$

$D \rightarrow 1CD \mid 1DG \mid \varepsilon$

$F \rightarrow 0FC \mid 0GF$

$G \rightarrow 0FD \mid 0GG \mid \varepsilon$

* CFL to CFG (Generate context free grammar from CFL)

1) $a^n b^n, n \geq 0$

$S \rightarrow aSb \mid \varepsilon$

2) $a^n b^{n+2}, n \geq 0$

$S \rightarrow aSb \mid bb$

3) $a^{2n} b^n, n \geq 0$

$S \rightarrow aaSb \mid \varepsilon$

4) $a^{2n+3} b^n, n \geq 0$

$S \rightarrow aaSb \mid aaa$

(5) $a^n b^n, n \geq 1$

$S \rightarrow aSb \mid ab$

(6) CFG for a language containing odd length string.

$S \rightarrow aX \mid bX$

$X \rightarrow aS \mid bS \mid \varepsilon$

1

PUMPING LEMMA FOR CFL.

pumping lemma for CFL is used to prove that a language is not context free.

Theorem:

Let 'L' be a CFL. Then there exists a constant n such that if z is any string in L such that |z| is atleast 'n' then we can write z = uvwxy (5 parts) subject to the following conditions.

1. $|vwx| \leq n$, That is the middle portion is not too long.

2. $vx \neq \varepsilon$. since v and x are the pieces to be pumped, this condition says that at least one of the strings we pump must not be empty.

3. For all $i \geq 0$, uv^iwx^iy is in L. That is the two strings v and x may be pumped any number of times, including 0 and the resulting string will still be a member of L.

pumping lemma for CFL is same as that of regular language the only difference is in pumping lemma for CFL

(3) Here 3 a's are followed by 5 'b's.
Hence the language is not CFL.
[our language consists of equal no. of a's followed by equal number of b's and equal number of c's]

Eg 2:

consider $L = \{ ww \mid w \in \{0,1\}^* \}$. prove L is not context free.

our strings are of the form.

0101

011011 , 010010, . . .

$L = \{ 00, 0101, 1111, 010010, 011011, \ldots \}$

Eg: string : $\underset{u}{0}\,\underset{v}{1}\,\underset{w}{1}\,\underset{x}{0}\,\underset{y}{11}$

Here I am taking pumping length $n = 3$

1) $|vwx| \leq 3$

2) $vx \neq \varepsilon$

now check the 3rd condition.

uv^iwx^iy for $i = 2$.

(4) $01^2 1 0^2 11$

$\Longrightarrow 0111\,0011 \notin L$

The above string is not in our CFL. Hence we can prove that L is not context-free.

Applications of pumping lemma (CFL)

The main application of pumping lemma for CFL is to prove that a language is not context free.

* CONTEXT SENSITIVE GRAMMAR (CSG)

A grammar $G = (V, T, P, S)$ is said to be context sensitive if all productions has the form

$$\alpha \longrightarrow \beta \quad \text{with} \quad |\alpha| \le |\beta|$$

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

The term context-sensitive comes from a normal form for these grammars, where the production is of the form "$\alpha_1 A \alpha_2 \longrightarrow \alpha_1 \beta \alpha_2$"

with $\beta \neq \varepsilon$.

(2) we break the string $z = uvwxy$ as 5 parts where as in regular language we break the string as $w = xyz$ as 3 parts.

Eg1:

prove that the language $L = \{a^n b^n c^n \mid n \geq 1\}$ is not context free.

$L = \{abc, aabbcc, aaabbbccc, \ldots\}$

Lets take our pumping length as 3 $(n=3)$

$$a^3 b^3 c^3 \Rightarrow \underbrace{aaa}_{u}\ \underset{v}{b}\ \underset{w}{b}\ \underset{x}{b}\ \underbrace{ccc}_{z}$$

here lets take $v = b$

$w = b$

$x = b$

1) $|vwx| \leq 3$

2) $vx \neq \varepsilon$.

now we have to check the third condition.

uv^iwx^iy when $i = 2$

$aaab^2bb^2ccc$

$\longrightarrow aaabbbbbbccc \notin L$

MODULE V

* CONTEXT SENSITIVE GRAMMAR (CSG)

A grammar $G = (V, T, P, S)$ is said to be context sensitive if all productions has the form

$$\alpha \longrightarrow \beta \quad \text{with} \quad |\alpha| \le |\beta|$$

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

The term context-sensitive comes from a normal form for these grammars, where the production is of the form " $\alpha_1 A \alpha_2 \longrightarrow \alpha_1 \beta \alpha_2$ "

with $\beta \neq \varepsilon$.

(3) They permit replacement of a variable A by string β only in the context $\alpha_1 - \alpha_2$.

* context sensitive grammars are more powerful than context free grammars because there are some languages that can be described by CSGs but not by CFGs.

The languages generated by these grammars are recognized by a linear bounded automata (LBA).

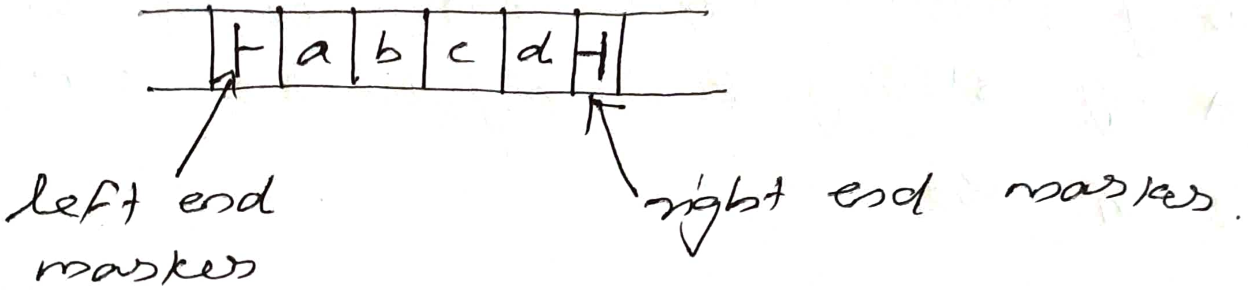
* Linear Bounded Automata (LBA).

Linear bounded Automata (LBA)

* A Linear bounded Automata Turing machine TM, is a variant of Turing machine which is created by restring the way in which the tape can be used.

* A Turing machine that uses only the tape space occupied by the input is called a linear bounded automata.

In LBA the tape head is not permitted to move off the portion of the tape containing the input. Input is restricted by using special symbols the left end marker ⊢ and the right end marker.



Tuple representation:

$$(Q, \Sigma, \Gamma, B, \delta, \vdash, \dashv, q_0, F)$$

This is the 9-tuple representation of LBA.

$Q \rightarrow$ set of states.

$\Sigma \rightarrow$ set of input symbols.

$\Gamma \rightarrow$ tape alphabet

$B \rightarrow$ Blank symbol.

$\vdash \rightarrow$ left end marker.

$\dashv \rightarrow$ right end marker.

$\delta \rightarrow$ Transition function

$q_0 \rightarrow$ starting state

$F \rightarrow$ final states (set of final states).

$$\delta : Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L, R\}}$$

Context Sensitive Languages:

The language that can be defined by

(1) context sensitive grammar is called CSL.

properties:

CSL are closed under union, intersection and complement.

Eg:

$S \rightarrow abc \mid aAbc$

$Ab \rightarrow bA$

$Ac \rightarrow Bbcc$

$bB \rightarrow Bb$

$aB \rightarrow aa \mid aaA$.

$S \Rightarrow aAbc$

$\Rightarrow abAc$ $(Ab \rightarrow bA)$

$\Rightarrow abBbcc$ $(Ac \rightarrow Bbcc)$

$\Rightarrow aBbbcc$ $(bB \rightarrow Bb)$

$\Rightarrow aaAbbcc$ $(aB \rightarrow aaA)$

$\Rightarrow aabAbcc$ $(Ab \rightarrow bA)$

$\Rightarrow aabbAcc$ $(Ab \rightarrow bA)$

$\Rightarrow aabbBbccc$ $(Ac \rightarrow Bbcc)$

$\Rightarrow aabBbbccc$ $(bB \rightarrow Bb)$

$\Rightarrow aaBbbbccc$ $(bB \rightarrow Bb)$

$\Rightarrow aaabbbccc$ $(aB \rightarrow aa)$

So the language is 3as followed by

3 b's and 3 c's. So the language generated by this Grammar is

$$L = \{ a^n b^n c^n \mid n \geq 1 \}.$$

* UNRESTRICTED GRAMMAR (TYPE 0 or Recursively enumerable)

A Grammar $G = (V, T, P, S)$ is called unrestricted, if all productions are of the form $\alpha \rightarrow \beta$ where α is a string of terminals and non-terminals with at least one non-terminal and α can not be null, β is a string of terminals and non-terminals.

ie, $\alpha \longrightarrow \beta$

where α is $(V+T)^* V (V+T)^*$

and β is $(V+T)^*$.

The languages generated by unrestricted grammars are Recursively Enumerable languages (REL).

Recursively Enumerable languages are Accepted (recognized) by a

(9) Turing machine.

* Eg: $Sab \to ba$
$A \to S$.
where S, A are variables and a, b are Terminals.

* TURING MACHINES

Turing machine was proposed by "Alan Turing" in 1936.
It is similar to finite automaton but with an unlimited and unrestricted memory.
*) A Turing machine is a much more accurate model of a general purpose computer.
A Turing machine can do everything that a real computer can do. None the -less, even a Turing machine can not solve certain problems. In a very real sense, these problems are beyond the theoretical limits of computation.
The Turing machine model uses an infinite tape as its unlimited memory. It has a tape head that can read and write symbols and move

around on the tape.

Notation for the turing machine.

A turing machine consists of a finite control, which can be in any of the states. There is a tape divided into cells, each cell can hold any one of the finite numbers of cells.

	B	B	x_1	x_2	...	x_i	...	x_n	B	B

Initially, the input, which is a finite-length string of symbols chosen from the input alphabet is placed on the tape. All other tape cells, extending infinitely to the left and right, initially hold a special symbol called the blank.
The blank is a tape symbol but not an input symbol.
There is a tape-head that is always positioned at one of the tape cells.
Initially the tape head is at the left most cell that holds the input. Here the tape head can read and write to the cell it is pointed at.

In a move the turing machine will

1. change the state. The next state optionally may be the same as current state.

2. Write a tape symbol in the cell scanned. The tape symbol replaces whatever symbol was in that cell. optionally the symbol written may be the same as the symbol currently there.

3. Move the tape head left or right.

Formal definition:

A Turing machine is a 7-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

$Q \rightarrow$ Finite set of states.

$\Sigma \rightarrow$ Finite set of input symbols.

$\Gamma \rightarrow$ set of tape symbols. Σ is a subset of Γ.

$q_0 \rightarrow$ start state.

$B \rightarrow$ The blank symbol. This symbol is in Γ but not in Σ.

$F \rightarrow$ set of final states.

(12)

$\delta \rightarrow$ The transition function.

$\delta(q, x) = (p, y, D)$

Here p is the state to which the transition takes place on input x. Thus x is replaced with y and D is the direction. It can be either R (Right) or L (Left).

Eg1: (Turing machines As Language Acceptors)

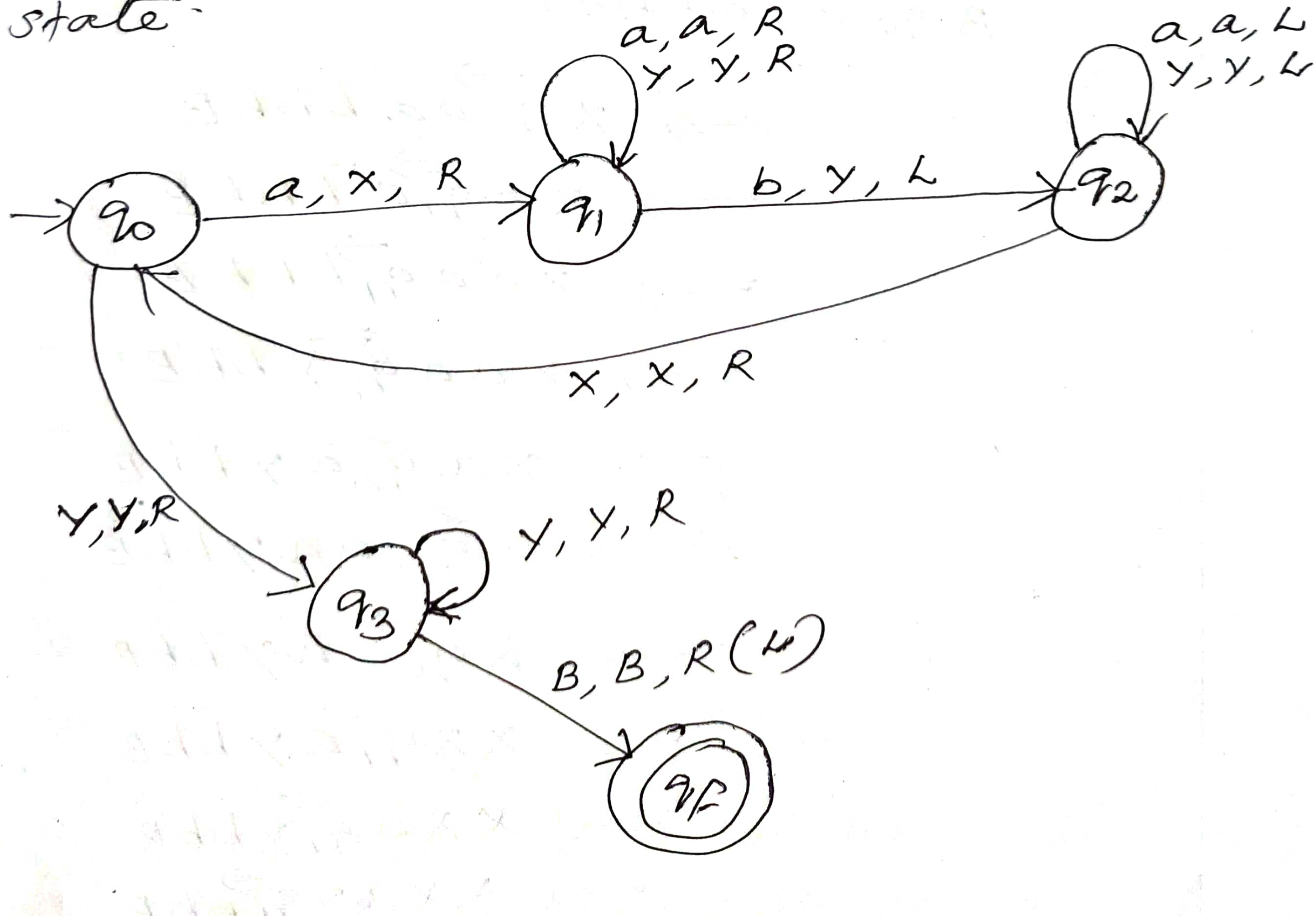
Design a turing machine accepting

$L = \{ a^n b^n \mid n \geq 1 \}$

$L = \{ ab, aabb, aaabbb \ldots \}$

Here the idea is when the turing machine gets the first input as 'a', then it replace it with an 'x' to remember that it has read that particular 'a'. Then moves the tape head in right direction keeping the symbol it scans as it is, until it gets the leftmost 'b', then to remember it replace 'b' with Y. and moves the tape head in left direction keeping the symbols it scans as it is till it reaches X, on getting X, it moves the tape head one position right and repeat the above cycle if it gets 'a'.

if it gets a Y instead of a, then it moves its tape head to the right keeping the symbol scanned as it is, as long as the symbol is Y, after skipping over all 'Y's in this way if the next symbol is 'B' then the string is accepted. Turing machine enters into the final state.



(14)

Transition Table.

	a	b	X	Y	B
q_0	q_1, X, R			q_3, Y, R	
q_1	q_1, a, R	q_2, Y, L		q_1, Y, R	
q_2	q_2, a, L		q_0, X, R	q_2, Y, L	
q_3				q_3, Y, R	$q_F, B, L/R$
q_F					

Instantaneous description.

i/p string : aaabbb.

$B\, q_0\, aaabbb\, B$

$\Rightarrow X\, q_1\, aabbbB$

$\Rightarrow X\, a\, q_1\, abbbB$

$\Rightarrow X\, aa\, q_1\, bbbB$

$\Rightarrow X\, a\, a\, q_2\, Y\, bbB$

$\Rightarrow X\, a\, q_2\, a\, Y\, bbB$

$\Rightarrow X\, q_2\, aaYbbB$

$\Rightarrow X\, q_0\, aaYbbB$

$\Rightarrow XX\, q_1\, aYbbB$

$\Rightarrow XXa\, q_1\, YbbB$

$\Rightarrow XXaY\, q_1\, bbB$

$\Rightarrow XXa\, Y\, q_2\, YbB$

$\Rightarrow xxa\overleftarrow{q_2}YYbB$

$\Rightarrow xx\overleftarrow{q_2}aYYbB$

$\Rightarrow xx\overrightarrow{q_0}aYYbB$

$\Rightarrow xxx\overrightarrow{q_1}YYbB$

$\Rightarrow xxxY\overrightarrow{q_1}YbB$

$\Rightarrow xxxYY\overrightarrow{q_1}bB$

$\Rightarrow xxxYY\overleftarrow{q_2}YB$

$\Rightarrow xxxY\overleftarrow{q_2}YYB$

$\Rightarrow xxx\overleftarrow{q_2}YYYB$

$\Rightarrow xxx\overrightarrow{q_0}YYYB$

$\Rightarrow xxxY\overrightarrow{q_3}YYB$

$\Rightarrow xxxYYq_3YB$

$\Rightarrow xxxYYY\overrightarrow{q_3B}$

$\Rightarrow xxxYYYq_f$.

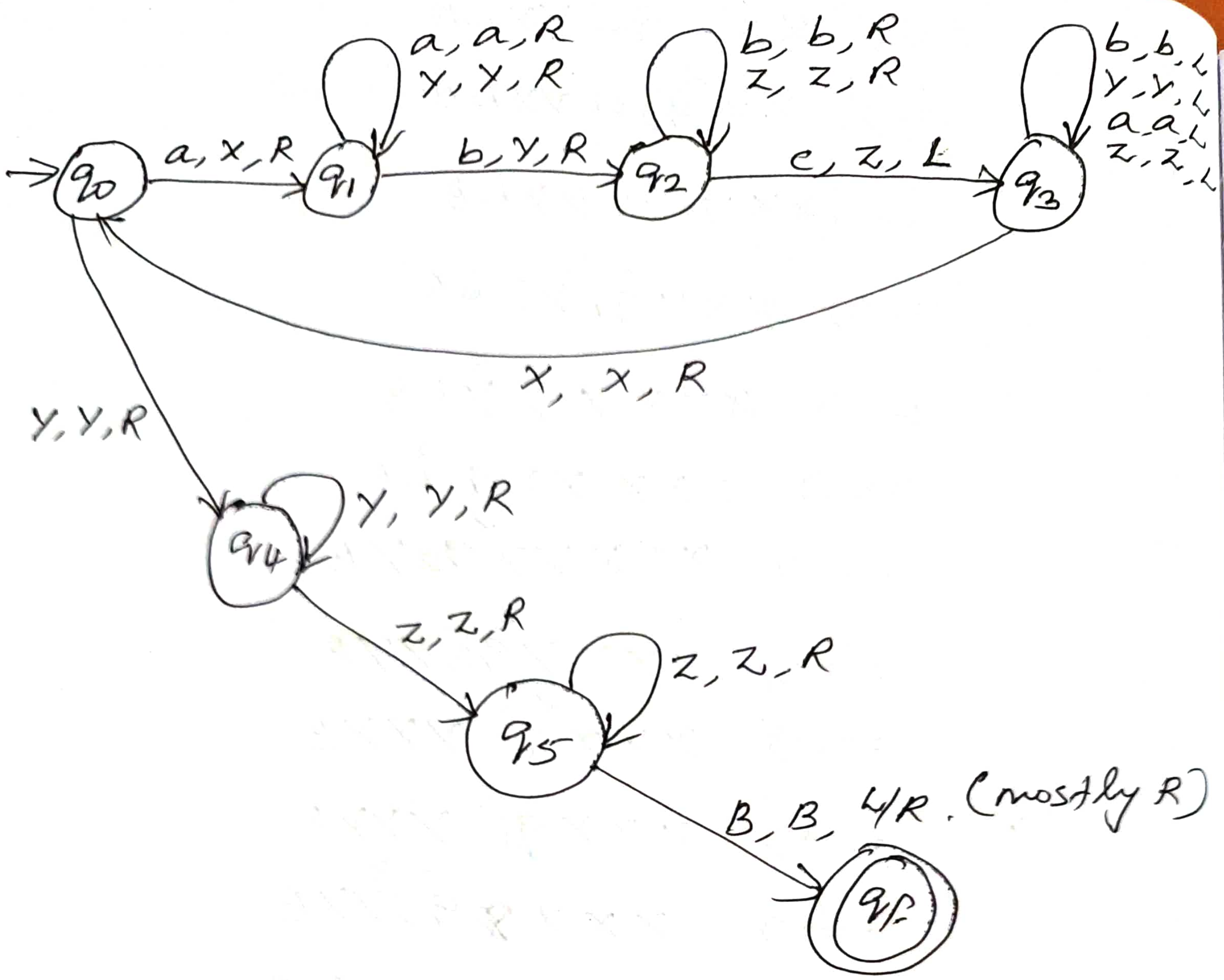
Eg 2:

Design a turing machine for the language $L = \{a^n b^n c^n \mid n \geq 1\}$.

$L = \{abc, aabbcc, \ldots\}$.

Here the idea is similar to the above problem. Here we need one more symbol to remember c.

(16)



Turing machine as transducer.

A Turing machine can be used as a transducer (as o/p is produced for a given i/p). The most obvious way to do this is to treat the entire non-blank portion of the initial tape as input and to treat the entire non-blank portion of the tape when the machine halts as output.

Eg 1:

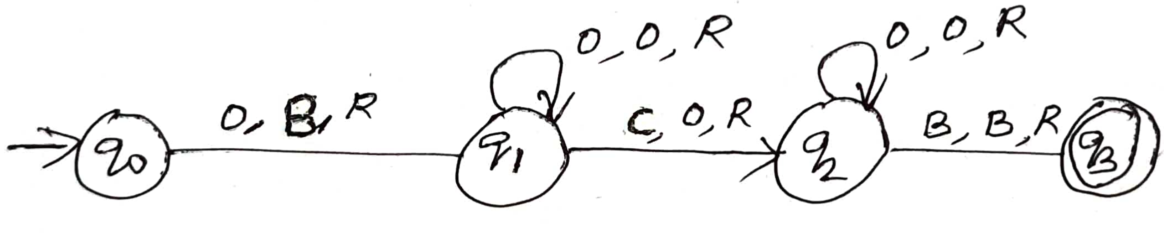
Design a TM to add 2 positive integers

For addition using turing machine unary format is followed. In unary format a number is represented by either all 1's or all 0's. For example 5 will be represented as 11111 or 00000. Here we use '0' for our representation.

For adding 2 numbers using a Turing machine both these numbers are given as input to the turing machine separated by a "c"

i/p : 00c000 (2+3)

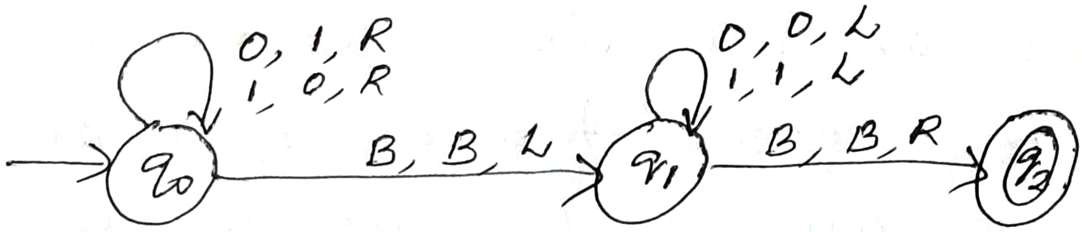
o/p : 00000



Here the first 0 is converted to blank B and the remaining 0s are kept as it is. once it gets a "c" that is also converted to 0 and the process continues till it gets a blank (B).

Eg 2:

Design a Turing Machine (TM) to find the 1's complement of a binary number



The procedure is to convert all 0's into 1's and all 1's into 0's. and goto right if B is found then goto left. Then ignore 0's and 1's and go left and if B is found then go to right.

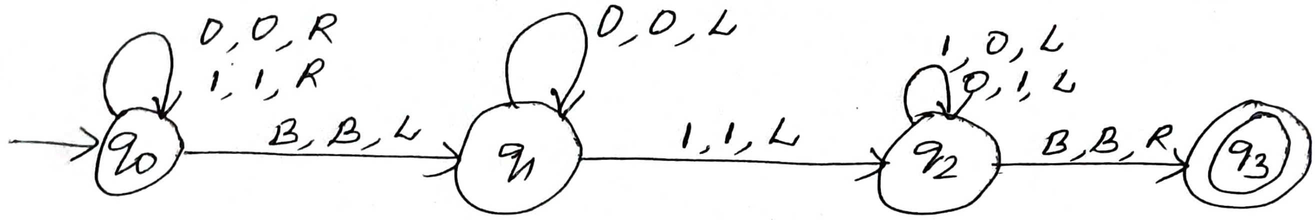
Eg 3:

Design a Turing machine (TM) to find the 2's complement of a binary number.

Here the procedure is

first ignore all 0's and 1's and go to right and then if B found goto left. Then ignore all 0's and go left if "1" found go to left. Then convert all 0's into 1's and all 1's into 0's and go to left & if B found go to right and stop the machine.

19



(1)

MODULE - VI

Variants of Turing Machine.

1) Multi tape turing machine
2) Non-deterministic Turing Machine
3) universal Turing Machine
4) Enumeration machine.

1) Multi tape turing machine:

A multi tape TM has a finite control (state) and some finite number of tapes.

Each tape is divided into cells and each cell can hold any symbol of the finite tape alphabet.

Initially,

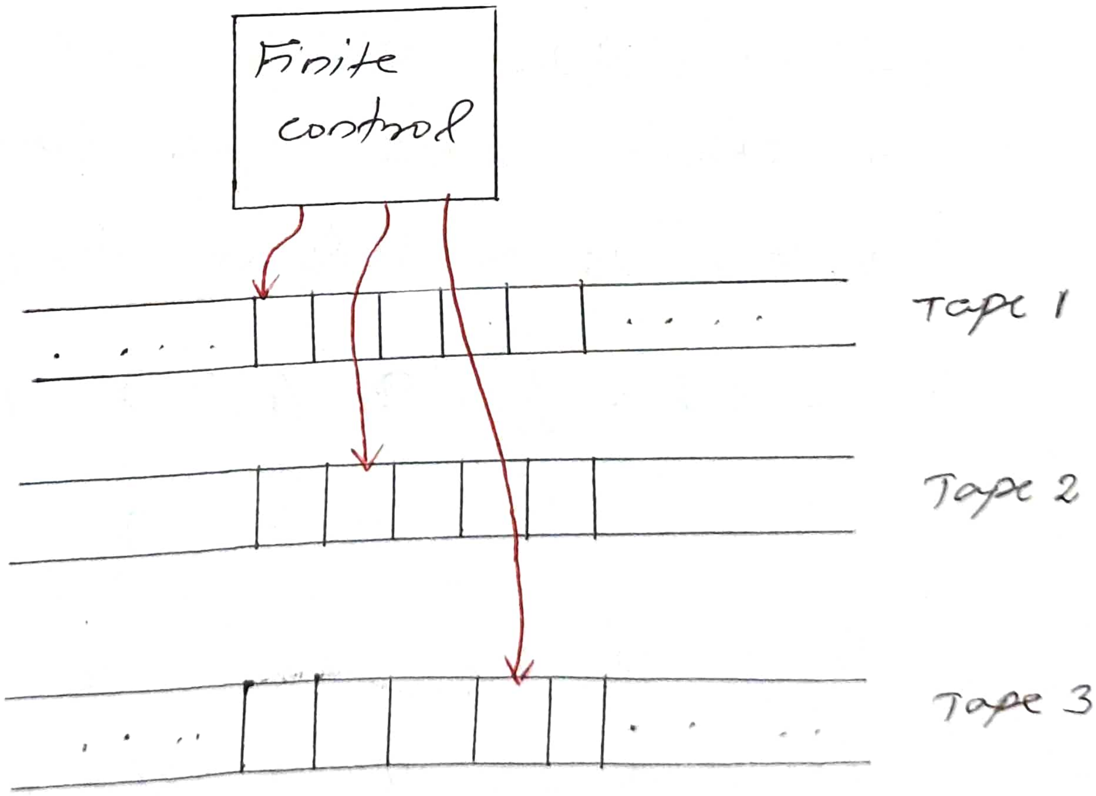
1. The input is placed on the first tape and all other cells of all the tapes hold the blank.
2. The finite control is in initial state and the head of the first tape is at the left end of the input.
3. All other tape heads are at some arbitrary cell.

2 In one move the multi-tape turing machine does the following.

1. The control enters a new state, which could be same as the previous state.

2. on each tape, a new symbol is written on the cell scanned. Any of these symbols may be the same as the previous symbol.

3. Each of the tape heads make a move, which can be either left, right or stationary.

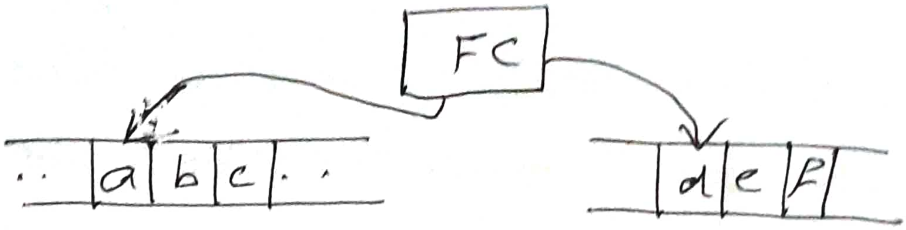


The tuple representation of Multi Tape TM is same as a single tape TM except the slight difference in "δ"

③

$\delta : Q \times \Gamma^{k} \longrightarrow Q \times \Gamma^{k} \times \{L, R\}^{k}$.

eg:



$\delta(q_2, a, d) = (q_3, x, y, L, R)$

2) Non-deterministic Turing Machine:

The tuple representation of a non-deterministic TM is same as deterministic TM, but it differs from deterministic TM in the transition function (δ), such that for each state and tape symbol X, $\delta(q, X)$ is a set of triples.

$$\delta(q, X) = \{ (q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots (q_k, Y_k, D_k) \}$$

Here $D_1, D_2 \ldots D_k$ are directions it can be either left or right.

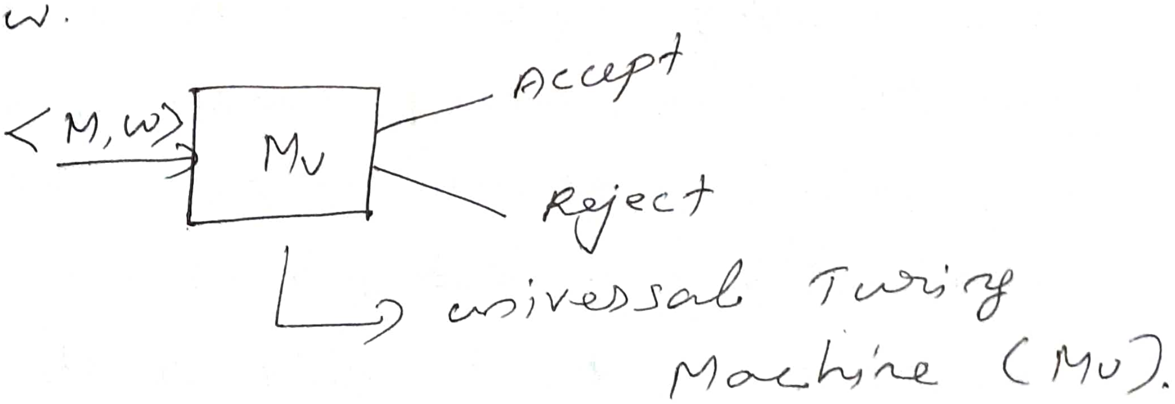
A non-deterministic TM can choose, at each step, any of the triples to be the next move.

$\delta : Q \times \Gamma \rightarrow$ power set of $(Q \times \Gamma \times \{L, R\})$

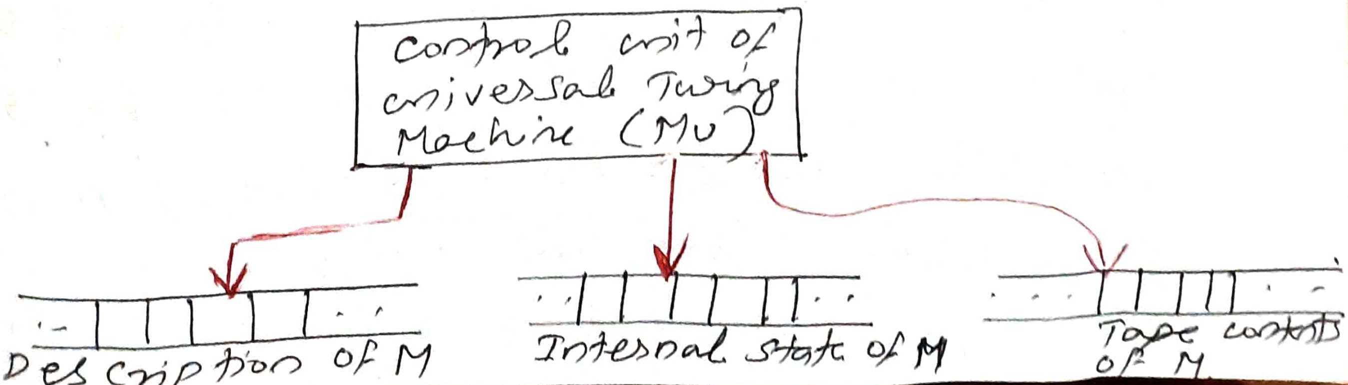
every non-deterministic TM has an equivalent deterministic TM. ie, the power of non-deterministic & deterministic TM are same.

3) <u>UNIVERSAL Turing Machine</u> (UTM)

A universal Turing machine is a turing machine that can simulate any other Turing machine. A UTM is an automaton that gives as input the description of any Turing machine M and string w, can simulate the computation of M on w.



A universal TM is a Multi tape TM shows as below.



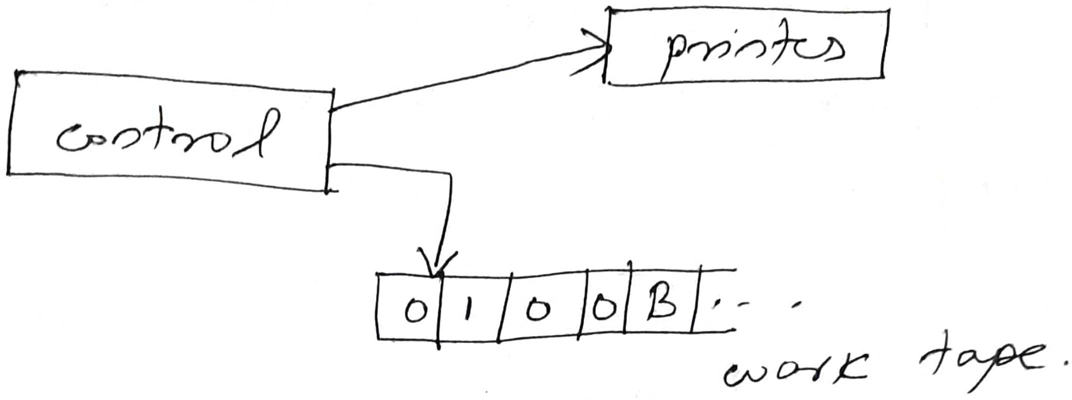
⑥ For any input M and w,
The first tape will contain the description of M in encoded form, the second tape will contain the internal state of M and the third tape will contain the tape contents of M. Mu looks first at the contents of tapes 2 and 3 to determine the configuration of M. it thus consults tape 1 to see what M would do in this configuration.

4) Enumeration machine.

An enumerator or enumeration machine is a turing machine with an attached printer. The turing machine can use the printer as an output device to print strings. Every time the turing machine wants to add a string to the list, it sends the string to the printer.

An enumerator E starts with a blank input on its work tape. If the enumerator doesnt halt it may print an infinite list of strings. The language enumerated by E is the

6 collection of all strings that it eventually prints out. E may generate the strings of the language in any order possibly with repetitions.



RECURSIVE and RECURSIVELY ENUMERA-BLE LANGUAGE

Recursive language (REC)

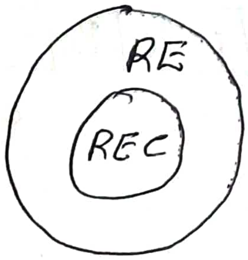
* it is also known as Turing decidable language.

A language L is said to be recursive such that there exists a turing machine M which accepts all the strings in L and thus it will halt and for the strings that are not in L, it will reject it and halt.

Recursively Enumerable language (RE)

* it is also known as Turing recognizable language.

(7) A Language L is said to be recursively Enumerable if there exists a Turing machine such that the TM will accept all the strings in L and halt. But for the strings that are not in L, it may or may not halt.



Recursive language is a subset of recursively enumerable language.

Properties of Recursive language.

1) union of two recursive language is also recursive. ie, recursive languages are closed under union.

2. The intersection of two recursive language is also recursive. ie, recursive languages are closed under intersection.

3. Recursive languages are closed under concatenation. ie, if L_1 and L_2 are 2 recursive languages

(8) then $L_1 \cdot L_2$ is also recursive.

4. The complement of a recursive language is also recursive. ie, recursive languages are closed under complement.

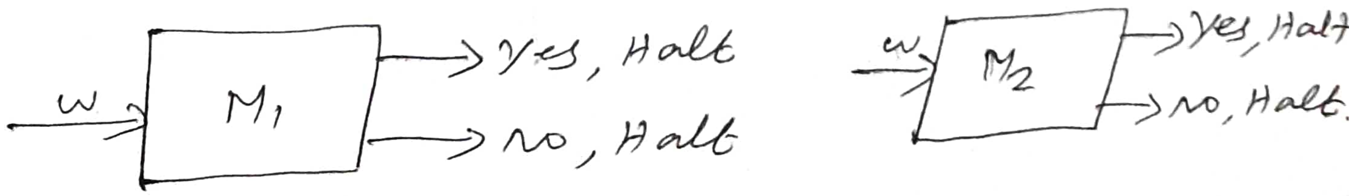
5. Recursive languages are closed under kleene closure (*). ie if L is recursive then L^* is also recursive.

proofs

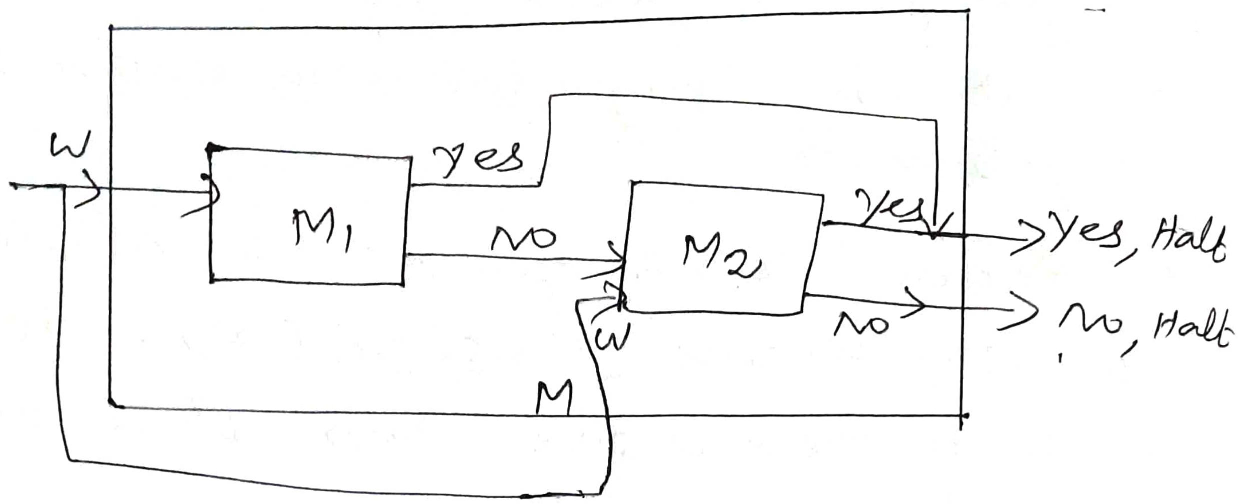
1) union of 2 recursive language is also recursive.

proof:

Let us consider 2 recursive languages L_1 and L_2 accepted by Turing machines M_1 and M_2. we construct a turing machine M which first simulates M_1. if M_1 accepts then M accepts. if M_1 rejects then M simulates M_2 and accepts if and only if M_2 accepts.



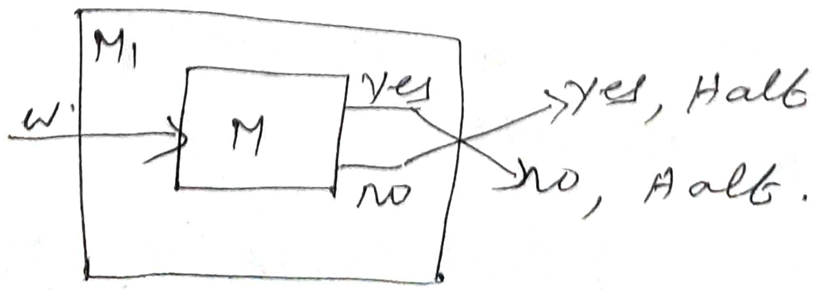
(9) The turing machine M accepts $L_1 \cup L_2$.



2) The complement of a recursive language is recurssive.

<u>proof:</u>

Let 'L' be a recurssive language and M be a turing machine that halts on all inputs and accepts L. Let us construct a turing Machine M_1 from M so that if M accepts w then M_1 rejects w and that if M rejects w then M_1 accept w.



(10) properties of recursively Enumerable Language.

1) union of 2 recursively Enumerable languages is also recursive.

2) Intersection of 2 recursively enumerable languages is also recursive.

3) Kleene closure of a recursive language is also recursive.

4) concatenation of 2 recursive languages is also recursive.

ie, Recursive languages are closed under union ($L1 \cup L2$), concatenation ($L_1 \cdot L_2$), intersection ($L_1 \cap L_2$), Kleene closure (L_1^*).

* Decidability and Halting problem.

In terms of a turing machine a problem is said to be decidable if there exists a corresponding turing machine which halts on every input with an answer yes or no. These problems are termed as turing decidable since a turing machine always halt on every i/p accepting or

(11) rejecting it.

Semi decidable or partially decidable problems are those for which a turing machine halts on the input accepted by it but it can either halt or loop forever on the input which is rejected by the turing machine.

Undecidable problems are problems for which we can't construct an algorithm that can answer the problem correctly in finite time are termed as undecidable problems. These problems may be partially decidable but never be decidable. That is there will always be a condition that will lead the turing machine into an infinite loop without providing an answer at all.

※ <u>Halting problem:</u>

It is undecidable to test whether an arbitrary turing machine will halt on an arbitrary input string.

12

Definition:

we define $H(M)$ for TM M to be the set of input strings w such that M halts on given input w, regardless of whether or not M accepts w. Thus the halting problem is the set of pairs (M, w) such that w is in $H(M)$.

Theorem:

There doesn't exist any TM H that behaves as the required definition of halting problem. The halting problem is therefore undecidable.

proof:

we assume that there exists an algorithm and consequently some Turing machine H that solves the halting problem. The input to H will be the string $w_M w$.

As required by definition, we want H to operate according to following rules.

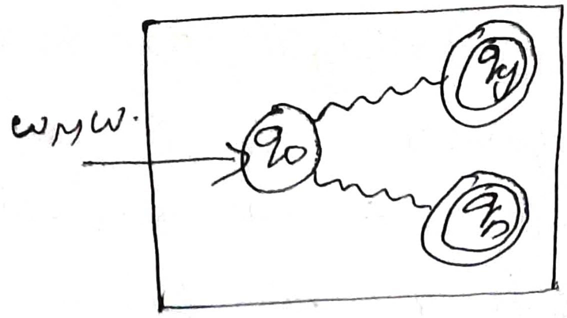
$$q_0 w_M w \vdash_H x_1 q_y x_2$$

if M applied to w halts

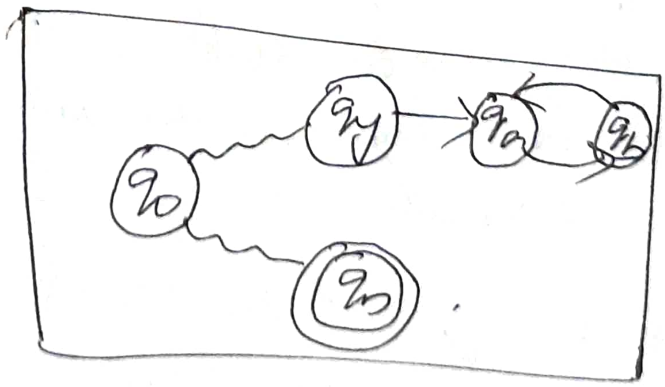
$q_0 w_M w \vdash_H y_1 q_n y_2$, if M applied

13

to w does not halt.



Next, we modify H to produce a turing machine H' with the structure shown in figure below. with the added states in figure, we want to convey that the transitions between state q_y and the new state q_a q_b are to be made, regardless of the tape symbol, in such a way that the tape remains unchanged.



comparing H and H' we see that in situations where H reaches q_y and Halts, the modified machine H' will enter an infinite loop.

(14)

The action of H' is described by

$$q_0 w_M w \vdash_{H'} \infty$$ if M applied to w halts, and

$$q_0 w_M w \vdash_{H'} y_1 q_n y_2$$, if M applied to w does not halt.

From H' we construct another turing machine H''. This new machine takes as input w_M and copies it, ending in its initial state q_0. After that it behaves exactly like H'. Then the action of H'' is such that

$$q_0 w_M \vdash_{H''} q_0 w_M w_M \vdash_{H''} \infty$$

if M applied to w_M does not Halt.

Now H'' is a turing machine, so it has a description $\{0,1\}^*$ say "$\hat{w}$". This string, in addition to being the description of H'', also can be used as input string.

$$q_0 \hat{w}'' \vdash_{H''} \infty$$ if H'' applied to $\hat{w}$ halts and

$$q_0 \hat{w}'' \vdash_{H''} y_1 q_n y_2$$, if H'' applied to $\hat{w}$

15

does not Halt. This is clearly nonsense.

The contradiction tells us that our assumption of existence of H, and hence the assumption of the decidability of the halting problem must be false.

Finite state Machines with output

Machines which can be formulated as FA with out is classified into two types

i) Moore Machine

ii) Mealy Machine

Moore Machine

It is the machine with finite number of states an for which, the output symbol depends only upon the present state of the machine.

The moore machine consists of 6-tuple

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

Q - Finite set of states.

Σ - input symbols or alphabet

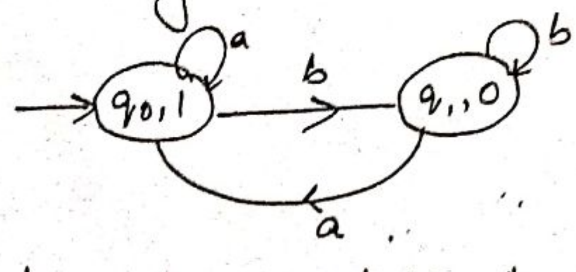
Δ - an output alphabet

λ - output function $(Q \to \Delta)$

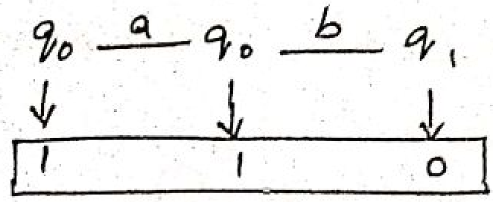
δ - transition function $(Q \times \Sigma \to Q)$

q_0 - starting state.

eg:



In the above moore machine, the state q_0 provides the output 1 and the state q_1 provides the output 0. for the input string ab in the machine, the traversal takes place as follows.



The output for the string ab is '110'

<u>Note :-</u>

The Moore machines produces $n+1$ output symbols for 'n' input symbols.

<u>Mealy Machine</u>

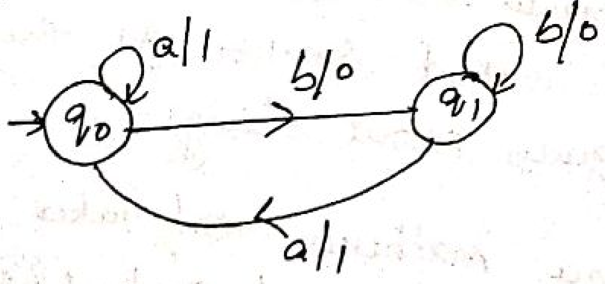
It is the machine with finite number of states and for which the output symbol is the function of the present input symbol as well as the present state of the machine.

A mealy machine is denoted by 6-tuple of the form

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

Q - set of states
Σ - input alphabet
Δ - output alphabet
δ - transition function $(Q \times \Sigma \rightarrow Q)$
λ - output function $(Q \times \Sigma \rightarrow \Delta)$
q_0 - initial state

eg :-



$\lambda : Q \times \Sigma \rightarrow \Delta$

$(q_0, a) \rightarrow 1$

$(q_0, b) \rightarrow 0$

$(q_1, a) \rightarrow 1$

$(q_1, b) \rightarrow 0$

for the i/p 'ab' the processing is carried out in the above mealy machine as follows.

$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1$

$\boxed{1 \quad 0}$

The output for the string ab is '10'

Note:-

The mealy machine produces 'n' output symbols for the 'n' input symbols.

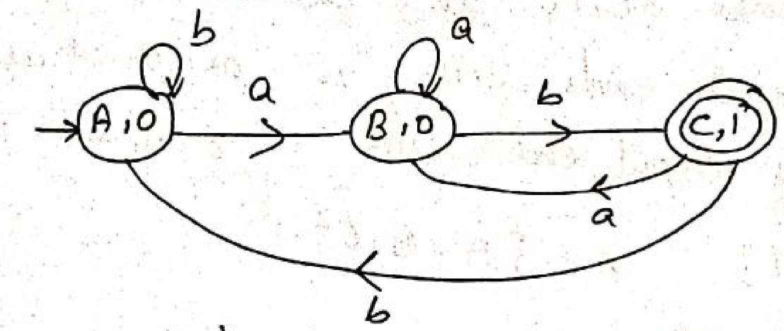
Difference between Moore and Mealy Machines:-

- Moore machines print character when in state.
- Mealy machines print character when traversing an arc.

Both moore and mealy machines has no final states. These machines provide some output only. In both Moore and mealy machines the process halts after producing the output symbol for the last input symbol of given input strings.

Q Construct Moore machines that takes set of all strings over {a,b} as input and print 1 as output for every occurrence of ab as a substring.

$\Sigma = \{a, b\}$ $\quad \Delta = \{0, 1\}$



for the input 'ab'

$A \xrightarrow{a} B \xrightarrow{b} C$

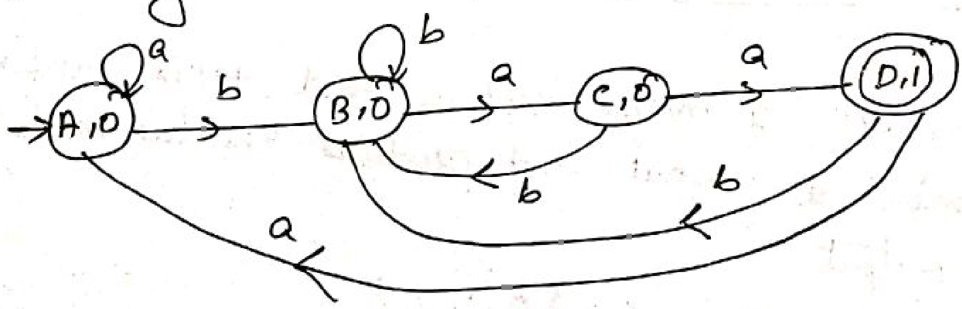
0	0	1

for the input abbbab

$A \xrightarrow{a} B \xrightarrow{b} C \xrightarrow{b} A \xrightarrow{b} A \xrightarrow{a} B \xrightarrow{b} C$

0	0	1	0	0	0	1

Q Construct a Moore Machine that takes set of all strings over $\{a, b\}$ as input and print 1 as output for every occurrence of baa as a substring



for the input baa the o/p is as follows

$A \xrightarrow{b} B \xrightarrow{a} C \xrightarrow{a} D$

0	0	0	1

for the input baabaa

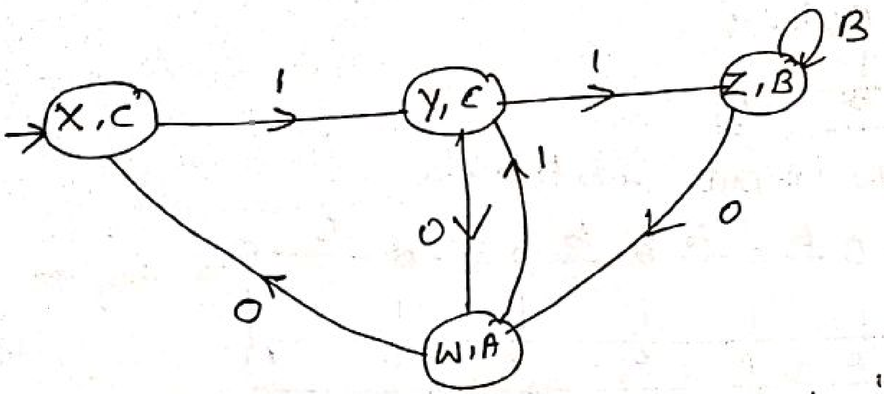
A — B — C — D — B — C — D

0	0	0	1	0	0	1

Q. Construct moore machine that takes set of all strings over {0,1} and produces 'A' as output if the input ends with '10' or produces 'B' as output if input ends with '11' otherwise produces 'C'.

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

10 — A
11 — B



for input '111'

$x \xrightarrow{1} y \xrightarrow{1} z \xrightarrow{1} z$

C	C	B	B

for input '110'

$x \xrightarrow{1} y \xrightarrow{1} z \xrightarrow{0} w$

C	C	B	A

Q. Construct a moore machine that takes binary number as input and produces residue module 3 as output.

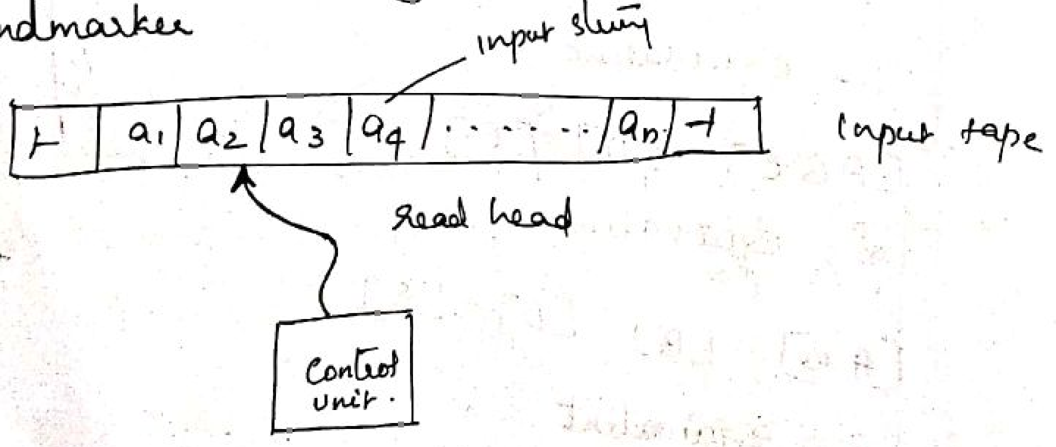
$\Sigma = \{0,1\}$ $\quad\quad \Delta = \{0,1,2\}$

	0	1	Δ
q_0	q_0	q_1	0
q_1	q_2	q_0	1
q_2	q_1	q_2	2

Two - Way Finite Automata

2DFA is a generalized version of the DFA which can revisit characters already processed i.e 2DFA can read the input back and forth with no limit

- 2DFA have a read head which can move left or right over the input string
- 2DFA consist of the symbols of the input string which is occupying in the cells of a finite tape one symbol per cell.
- The input symbol is enclosed in left and right endmarkers ⊢ and ⊣ which are not the elements of the input alphabet Σ
- The read head may not move outside of the endmarker



Formal Definition of 2DFA :-

A two-way deterministic finite automata (2DFA) is a quadruples $M = (Q, \Sigma, \delta, q_0, F)$ where

Q – finite set of states

Σ – finite set of input alphabet

q_0 – starting state

F – set of final states which is a subset of Q

δ is a transition function defined from $Q \times \Sigma$ to $Q \times \{L, R\}$

If $\delta(q, a) = (p, L)$ then in state q, seeing an input symbol 'a', the DFA enters into state 'p' and moves its head left one cell.

If $\delta(q, a) = (p, R)$ then in state q, seeing an input symbol 'a', the DFA enters the state 'p' and moves its head Right one cell

A string is said to be accepted by a 2DFA iff it reads off the right end of the tape and at the same time entering an accepting state.

eg:-
Consider the transition table given below. and check whether the string '101001' is accepted or not by the 2DFA

states	0	1
$\rightarrow q_0$	(q_0, R)	(q_1, R)
q_1^*	(q_1, R)	(q_2, L)
q_2	(q_0, R)	(q_2, L)

Acceptability of the string using 2DFA is given by

$q_0 101001 \vdash 1 q_1 01001$

$\vdash 10 q_1 1001$

$\vdash 1 q_2 01001$

$\vdash 10 q_0 1001$

$\vdash 101 q_1 001$

$\vdash 1010 q_1 01$

$\vdash 10100 q_1 1$

$\vdash 1010 q_2 01$

$\vdash 10100 q_0 1$

$\vdash 101001 q_1$

Since the tape reaches at the right end of the input string and enters into final state the given input string is accepted.

A Grammar G is a set of rewriting or

Decision problems Related With CFL's

i) Emptiness:

In a grammar G, L(G) is nonempty if and only if 'S', the starting symbol is useful, Otherwise L(G) is empty

ii) Finite:

Assume that G is in chomsky normal form with no unit productions, no useless symbols and no nullable symbols.

In CNF all productions are of the form

$A \rightarrow a$

or

$A \rightarrow BC$

Let 'H' be a graph with vertices V. Draw an arrow from $A \rightarrow B$ and $A \rightarrow C$ for each production $A \rightarrow BC$. Then L(G) is Finite if and only if H has no cycles.

iii) Infinite:-

Assume that G is in chomsky normal form with no unit productions, no useless symbols and no nullable symbols.

In CNF all productions are of the form

$A \rightarrow a$

or

$A \rightarrow BC$

Let H be the graph with vertices V. Draw an arrow from $A \to B$ and $A \to C$ for each production $A \to BC$. Then $L(G)$ is infinite if and only if it has atleast one cycle in a directed graph.

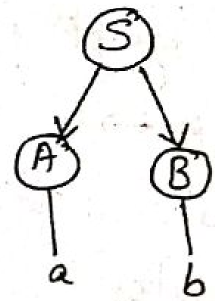
Example:-

Consider the grammar given below

$S \to AB$
$A \to BC/a$
$B \to CC/b$
$C \to AB/a$

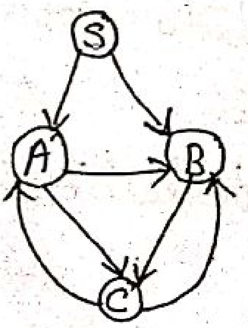
i) Check the above given grammar is empty or not for the input ab

$S \to AB \to aB \to ab$



ii) Check whether the above given grammar is infinite or not.

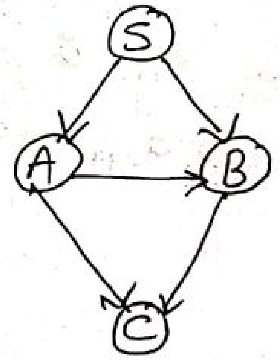
If the grammar is infinite then the directed graph contain the cycle.



Cycle – BCB
– ACA

iii) Check whether the grammar given below is finite or not

$S \rightarrow AB$
$A \rightarrow BC / a$
$B \rightarrow CC / b$
$C \rightarrow a$



The directed graph do not contain any cycle
So the given grammar is finite.

Membership Algorithm:-
CYK (Cocke - Younger - kasami) algorithm is used for finding whether the given string is a member of the given grammar or not.
The input to the CYK algorithm should be a Chomsky Normal Form that
CYK Algorithm starts with a CNF grammar $G=(V,T,P,S)$ for the language L. The input to the algorithm is a string $w = a_1 a_2 \ldots a_n$ in T^*.
The Complexity of the algorithm is $O(n^3)$ ie the

algorithm constructs a table that tells whether w is in L in $O(n^3)$ time

The construction of the triangular table is given below.

The horizontal axis corresponds to the positions of the string $w = a_1 a_2 \ldots a_n$ which have a length of 5. The table entry X_{ij} is the set of variables 'A' such that $A \overset{*}{\Rightarrow} a_i a_{i+1} \ldots a_j$

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
a_1	a_2	a_3	a_4	a_5

To fill the table, we are moving row-by-row upwards. Each row corresponds to one length of substring, the bottom row is for strings of length 1, the second from bottom row for strings of length 2, and so on, until the top row corresponds to the one substring of length n, which is 'w' itself.

CYK Algorithm

begin

1. for $i = 1$ to n do

2. $V_{i1} = \{ A \mid A \to a$ is a production and the i^{th} symbol of ~~x is~~ a $\}$

3. For $j = 2$ to n do

4. For $i = 1$ to $n-j+1$ do

begin

5. $V_{ij} = \phi$

6. For $k = 1$ to $j-1$ do

7. $V_{ij} = V_{ij} \cup \{ A \mid A \to BC$ is a production, B is in V_{ik} and C is in $V_{i+k, j-k} \}$

end.

end.

Q. Consider the following CFG,

$S \to AB \mid BC$
$A \to BA \mid a$
$B \to CC \mid b$
$C \to AB \mid a$

Check whether the input baaba is in the CFG or not.

To check whether the input is in CFG or not draw the triangular table in which the bottom row consists of five columns.

	S,A,C				
	ϕ	S,A,C			
	ϕ	B	B		
	A,S	B	S,C	A,S	
	B	A,C	A,C	B	A,C
	b	a	a	b	a

$t_{11} = B \; (B \to b)$

$t_{21} = A, C \; (A \to a, C \to a)$

$t_{31} = A, C \; (A \to a, C \to a)$

$t_{41} = B \; (B \to b)$

$t_{51} = A, C \; (A \to a, C \to a)$

t_{12} contain string of length 2.

ie t_{12} contain ba

b — B
a — A
  — C

$\therefore$ ba = (B A) or (B C)

$\therefore t_{12} = A, S \; (A \to BA, S \to BC)$

$t_{22} =$ a a

A C A C

AA, AC, CA, CC

ϕ ϕ ϕ B

$\therefore t_{22} = B$

$t_{32} =$ a b

A C B

AB CB

S, C ϕ

$\therefore t_{32} = SC$

$t_{42} = ba$

$b \quad a$

$B \quad AC$

$BA \quad BC$

$A \quad S$

$\therefore t_{42} = A, S$

$t_{13} = baa$

$\Downarrow$

$b \; aa$ or $ba \; a$

$t_{11} \quad t_{22} \qquad t_{12} \quad t_{31}$

$B \quad B \qquad AS \quad AC$

$BB \qquad AA \quad AC \quad SA \quad SC$

$\phi \qquad \phi \quad \phi \quad \phi \quad \phi$

$\therefore t_{13} = \phi$

$t_{23} = aab$

$\Downarrow$

$a \; ab \qquad aa \; b$

$t_{21} \quad t_{32} \qquad t_{22} \quad t_{41}$

$AC \quad SC \qquad B \quad B$

$AS \quad AC \quad CS \quad CC \quad BB$

$\phi \quad \phi \quad \phi \quad B \quad \phi$

$\therefore t_{23} = B$

$t_{33} = aba$

$\Downarrow$

$a \; ba \qquad ab \; a$

$t_{31} \quad t_{42} \qquad t_{32} \quad t_{51}$

$AC \quad AS \qquad SC \quad AC$

$AA \quad AS \quad CA \quad CS \quad SA \quad SC \quad CA \quad CC$

$\phi \quad \phi \quad \phi \quad \phi \quad \phi \quad \phi \quad \phi \quad B$

$\therefore t_{23} = B$

t_{14} - b a a b

b aab   ba ab   baa b

t_{11} t_{22} t_{12} t_{32} t_{13} t_{41}

B   B   AS   SC ϕ B

BB   AS AC SS SC ϕ

ϕ ϕ ϕ ϕ ϕ

$\therefore t_{14} = \phi$

t_{24} - a a b a

a aba   aa ba   aab a

t_{21} t_{33} t_{22} t_{42} t_{23} t_{51}

AC   B   B   AS   B   AC

AB CB   BA BS   BA BC

S, C ϕ A ϕ A   S

$\therefore t_{24} = S, A, C$

t_{15} = b a a b a

b aaba   ba aba   baa ba   baab a

t_{11} t_{24} t_{12} t_{33} t_{13} t_{42} t_{14} t_{51}

B   SAC   AS   B ϕ AS ϕ AC

BS BA BC   AB SB ϕ ϕ

ϕ A S   S, C ϕ

$\therefore t_{15} = S, A, C$

The topmost cell contain start symbol of CNF 'S'. So the given string baaba is in $L(G)$.