CSC 311: AUTOMATA THEORY, COMPUTABILITY AND FORMAL LANGUAGES

Course Development Team

Content Authoring

Bajeh A.O., Mojeed H.A.

Content Editor

Mrs. Bankole

Instructional Design

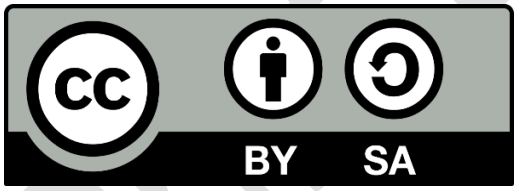
Miss. Damilola Adesodun

Dr. Olawale S. Koledafe

Mr. Samuel A. Adekanye

Published by the Centre for Open and Distance Learning,

University of Ilorin, Nigeria



Centre for Open and Distance Learning

University of Ilorin,

Nigeria

E-mail: codl@unilorin.edu.ng

Website: https://codl.unilorin.edu.ng

From the Vice Chancellor

Courseware development for instructional use by the Centre for Open and Distance Learning (CODL) has been achieved through the dedication of authors and the team involved in quality assurance based on the core values of the University of Ilorin. The availability, relevance and use of the courseware cannot be timelier than now that the whole world has to bring online education to the front burner. A necessary equipping for addressing some of the weaknesses of regular classroom teaching and learning has thus been achieved in this effort.

This basic course material is available in different electronic modes to ease access and use for the students. They are available on the University's website for download to students and others who have interest in learning from the contents. This is UNILORIN CODL's way of extending knowledge and promoting skills acquisition as open source to those who are interested. As expected, graduates of the University of Ilorin are equipped with requisite skills and competencies for excellence in life. That same expectation applies to all users of these learning materials.

Needless to say, that availability and delivery of the courseware to achieve expected CODL goals are of essence. Ultimate attention is paid to quality and excellence in these complementary processes of teaching and learning. Students are confident that they have the best available to them in every sense.

It is hoped that students will make the best use of these valuable course materials.

Professor S. A. Abdulkareem
Vice Chancellor

Foreword

Courseware is the livewire of open and distance learning. Whereas some institutions and tutors depend entirely on Open Educational Resources (OER), CODL at the University of Ilorin considered it necessary to develop its materials. Rich as OERs are and widely as they are deployed for supporting online education, adding to them in content and quality by individuals and institutions guarantees progress. Pursuing this goal has brought the best out of the Course Development Team across Faculties in the University. Despite giving attention to competing assignments within their work setting, the team has created time and eventually delivered.

The development of the courseware is similar in many ways to the experience of a pregnant mother eagerly looking forward to the delivery date. As with the eagerness for a coming baby, great expectation pervaded the air from the University Administration, CODL, Faculty and the writers themselves. Careful attention to quality standards, ODL compliance and UNILORIN CODL House Style brought the best out from the course development team. Response to quality assurance with respect to writing, subject matter content, language and instructional design by the authors, reviewers, editors and designers, though painstaking, has yielded the course materials now made available primarily to CODL students as open resources.

Aiming at parity of standards and esteem with regular university programmes is usually an expectation from students on open and distance education programmes. The reason being that stakeholders hold the view that graduates of face-to-face teaching and learning are superior to those exposed to online education. CODL has the dual mode mandate. This implies a combination of face-to-face with open and distance education. With this in mind, the Centre has developed its courseware to combine the strength of both modes to bring out the best from the students. CODL students and other categories of students of the University of Ilorin and similar institutions will find the courseware to be their most dependable companion for the acquisition of knowledge, skills and competences in the respective courses and programmes.

Activities, assessments, assignments, exercises, reports, discussions and projects at various points in the courseware are targeted at achieving the objectives of teaching ang learning. The courseware is interactive and directly points the attention of students and users to key issues helpful to their particular learning. The student's understanding has been viewed as a necessary ingredient at every point. Each course has also been broken into modules and their component units in an ordered sequence. In it all, developers look forward to successful completion by CODL students.

Courseware for the Bachelor of Science in Computer Science housed primarily in the Faculty of Communication and Information Science provide the foundational model for Open and Distance Learning in the Centre for Open and Distance Learning at the University of Ilorin.

Henry O. Owolabi
Director, CODL

Course Guide

CSC 311 is titled Automata Theory and it is a case taken in the third year of the Computer Science programme. CSC 311 is a 3-unit and compulsory course for students majoring in Computer Science. Thus, it must be taken and passed.

The main objective of the course is to teach the fundamental concepts of automation of processes and systems. Speicifically, the concepts of Finite state automaton, push down automation and turing machine, and the theories underlying automatons are explained in-depth with examples. Also, Grammar and formal languages are taught to establish the fundamentals of how programming languages are developed.

The course is organized into five (5) distinct modules with each module addressing a major component of the course and made up of two or more units.

Course Goal

The goal of this course is to prepare students with a strong knowledge of automation and the underlying theories, and also to teach them the fundamental concepts that form the basis of formal languages and grammar. This will prepare the student for higher level courses such as organization of programming languages which deals with how computer programming languages are developed for computation.

Related Courses

Prerequisite:

CSC 111 – Introduction to Computing I

CSC 212 – Computer Programming II

Required for:

CSC 214 – Introduction to File Processing. Credits (C)2.

CSC 226 – Computer Appreciation III. Credits (E) 2.

Learning Outcomes

At the end of this course, you should be able to:

i. Describe the concept of automata;

ii. Mention the different types of computational models available;

iii. Differentiate among the different Computer models and state their unique computational power;

iv. Model computation problems using the appropriate computation models;

v. Describe formal grammars and identify the types of grammar;

vi. Develop formal grammars given the definition of languages; and

vii. Perform validation and verification of given formal language sentences using formal grammars.

Table of Contents

Requirements for Success

The CODL Programme is designed for learners who are absent from the lecturer in time and space. Therefore, you should refer to your Student Handbook, available on the website and in hard copy form, to get information on the procedure of distance/e-learning. You can contact the CODL helpdesk which is available 24/7 for every of your enquiry.

Visit CODL virtual classroom on http://codllms.unilorin.edu.ng. Then, log in with your credentials and click on CSC 222. Download and read through the unit of instruction for each week before the scheduled time of interaction with the course tutor/facilitator. You should also download and watch the relevant video and listen to the podcast so that you will understand and follow the course facilitator.

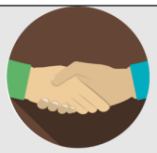
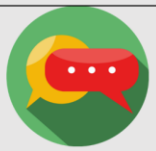
At the scheduled time, you are expected to log in to the classroom for interaction.

Self-assessment component of the courseware is available as exercises to help you learn and master the content you have gone through.

You are to answer the Tutor Marked Assignment (TMA) for each unit and submit for assessment.

Embedded Support Devices

Throughout your interaction with this course material, you will notice some set of icons used for easier navigation of this course materials. We advise that you familiarize yourself with each of these icons as they will help you in no small ways in achieving success and easy completion of this course. Find in the table below, the complete icon set and their meaning.

		
Introduction	Learning Outcomes	Main Content
		
Summary	Tutor Marked Assignment	Self-Assessment Question
		
Web Resources	Downloadable Resources	Discuss with Collegues
		
References	Further Reading	Self-Exploration

Assessment and Grading

TMA	20%
Continuous Assessment (CA)	20%
Examination	60%
Total	100%

Module 1

Unit 1: Computation and Computability theory

Unit 2: Introduction to Automata theory

Unit 1: Computation and Computability Theory

1.0 Introduction

This unit introduces you to the theory of automata and its associated concept of computation and computability. It forms the foundation on which the rest of the course depends.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

i. Explain the concepts of automata, computation and computability
ii. Formally define an automata
iii. Depict finite systems as diagraphs called state machines

3.0 Main Content

3.1 Introduction

Automata theory is the branch of theoretical Computer Science that deal with the logic of computation using simple machines called automata, which are abstract representation of computing machines that accept certain input and moves through one or more states to yield outputs. Automata theory provides a platform through which Computer Scientists can study the dynamic nature of discrete systems which include three main parts: inputs, outputs and states. The inputs and outputs are finite sets of finite sequence of symbols, and the states are finite number of points that the machine can attain due to some input.

3.2 Computation

In theoretical computer science, computation refers to a mapping that associates to each element of an input set X exactly one element in an output set Y.

Using the following symbols: {0,1}, the set of all possible combination of these symbols, denoted as {0,1}*, is the Kleen closure or Kleen plus. Computation deals mainly with decision problems of the type "given an integer n, decide whether or not n is a prime number rather than equivalent problem of the form" "given two real numbers x and y, what is their product $x \times y^n$.

Definition 1.0: a decision problem is a mapping

$$\{0, 1\}^* \longrightarrow \{0, 1\}$$

That takes is input any finite strings of 0's and 1's and assign to it an output consisting of either 0(No) or 1(Yes).

3.3 Computability

A common problem in the field of theoretical computer science is the theory of what can or cannot be solved or computed. Many problems which appear easy at first glare are impossible to solve in generals. Consequently, the first step in solving this problem is to decide precisely if the problem is computable.

What does it mean for a function or problem to be computable? The answer to these question forms the basis of computability theory. According to Church-turning thesis, computable function are exactly the functions that can be calculated using a mechanical calculating device given unlimited amount of time and storage space. This thesis also states that anytime which has an algorithm is computable i.e. the computability of a problem is closely linked to the existence of an algorithm to solve the problem. Thus, for a function or problem to be computable, it must have an algorithm which will expect some input, perform some calculations and it terminates, return a unique result as output.

Several computational models or machines have been invented in order to solve the problem of computability. The study of these computational machines is the focus of this course.

Unit 2: Introduction to Automata

1.0 Introduction

In this unit, we will delve into the essential concepts of automata, computation, and computability, establishing a framework for understanding how these elements interact within discrete systems. We will explore the structure of automata, including their inputs, outputs, and states, and how these components work together to form a coherent model of computation

2.0 Learning Outcomes

At the end of this unit, you should be able to:

i. Define the key concepts of automata
ii. Illustrate the structure and functioning of various types of automata
iii. Analyze different computational models

3.0 Main Content

3.1 Introduction

An Automaton (plural Automata) is an abstract computing machine or device that successively reads each symbols of an input string, and changes its state according to a particular control

mechanism. By abstract machine it means a device need not to be physical hardware! A computing device meaning must possess a mechanical system that model a procedure or an algorithm. Thus, when given a set of inputs, an automaton runs automatically without direct involvement of man.

Examples of automata are vending machine, turnstile, printing machine, PC, packing ticket machine, etc.

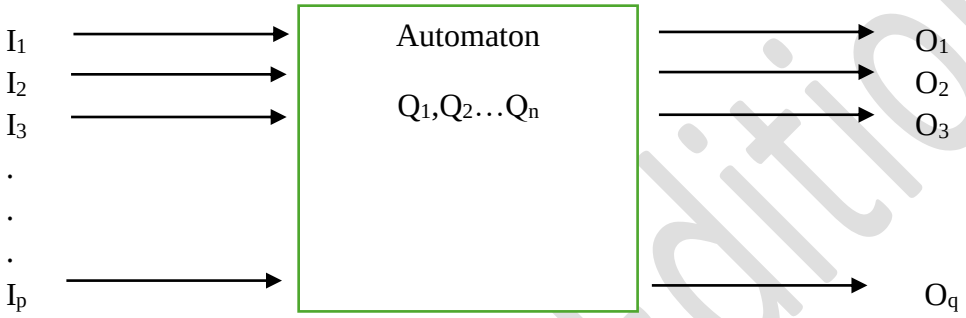


Figure 1.1: Modal of an Automaton

Characteristics of an Automaton

i. Input: $I_1,I_2,I_3…I_p$ are the input of the automaton each of which can take a finite number of fixed values.
ii. Output: $O_1,O_2,O_3…O_q$ are the outputs of the automaton each of which can take a finite number of fixed values.
iii. States: at any instant of time the automaton can be in any of the state i.e. $Q_1,Q_2,Q_3…Q_n$.
iv. State Relation: the next state of an automaton at any instance of time determined by the present state and present input. (also called Transition)
v. Output Relation: the output of an automaton is determined by either states only or both the input and the state or input only.

An Analogy of a turnstile is described below.
A Turnstile is a machine viewed as a rotating gate with a coin receiver, it is composed of two states: lock and unlock state. The gate can either be locked or unlocked based on the input to the machine. The gate is unlocked as the receiver receives a coin. The Turnstile does not rotate after

receiving the coin until a push on the turnstile is realized, and the gate is locked until another coin is received. The diagram below represents the possible states of the turnstile.

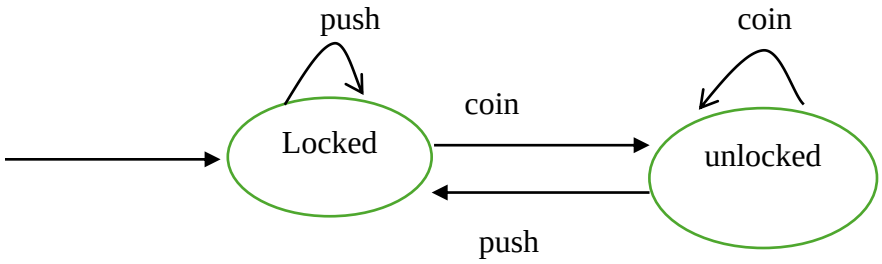


Figure 1.2 A model of a Turnstile

Below is a tabular representation of the state transition of the Turnstile.

Table 1.1: state transition function for Turnstile:

Current state	**Input**	**Next state**	**Output**
Locked	Coin	Unlocked	Unlocked turnstile
	Push	Locked	None
Unlocked	Coin	Unlocked	None
	Push	Locked	Locked turnstile

3.5 Types of Automata

There are three main types of automata namely:

i. Finite state machine (FSM)
ii. Push down automata
iii. Turing machine

3.5.1 Finite State Machine (FSM)

FSM are machines with finite and limited states which serves as memory for the machine. It is composed of an input tape, reading head, finite control and optionally an output tape. The particular characteristic of these machines is the finite memory which means they have considerably low memory or limited memory and therefore less powerful. Figure 1.3 depicts the architecture of FSM.

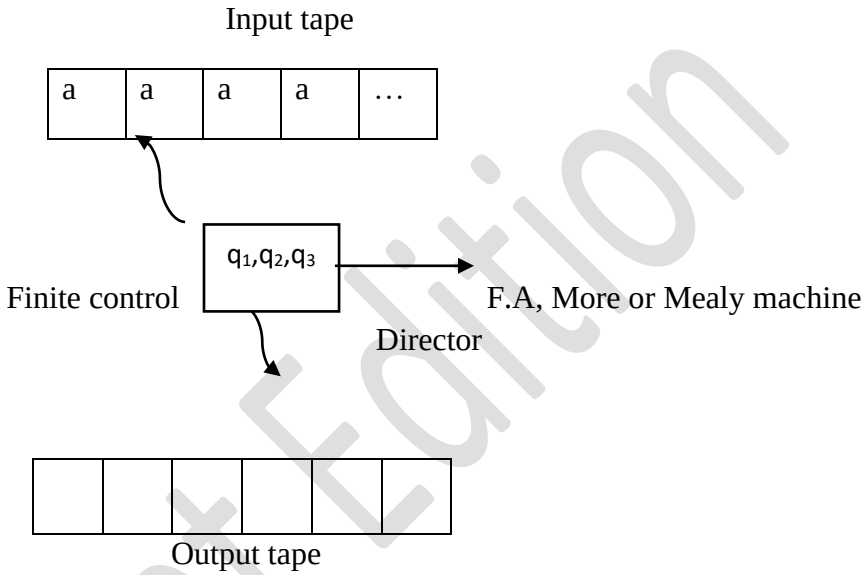


Figure 1.3: Finite State Machines (FSM)

FSM are of three types:

i. Finite State Automata (FSA) or Finite Automata (FA): an FSM with only input tape, reading head and finite control (system of states). It comprises of DFA and NPA; these machines has the power to recognize pattern with no output tape.

ii. Moore Machines: an FA with an additional output tape and a writing head. Here, the output is dependent only on the states. This machines can recognize and count.

iii. Mealy Machines: an FA with input and output tape as well as reading and writing head as in Moore machines but here, the output departs on both the state and the input; these machines can recognize and compute result.

3.5.2 Pushdown Automata

A pushdown automaton is a kind of automaton that make use of a stack. As the stack structured provide additional infinite memory for the machine and therefore pushdown automaton is more powerful than a finite state machine. It is composed of input tape, a reading head, finite control and a stack. The top of the stack is manipulated as part of the processing and calculation of the machine.

A

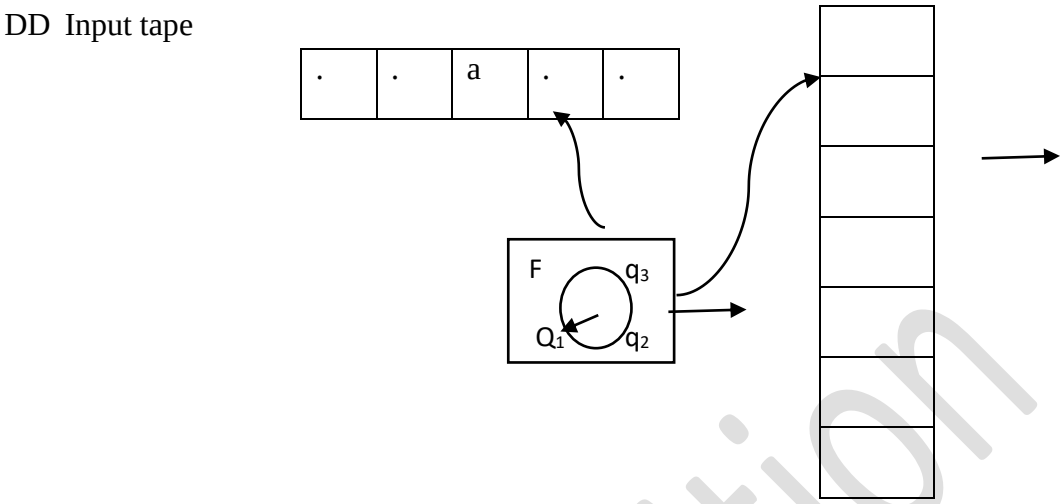


Figure 1.4: An illustration of a push down automata

These machines has the power to parse syntax, math pattern and evaluate functions.

3.5.3 Turning Machine

This class of automata represent the most powerful automata. It is composed of infinite input tape, left or right and a head that is capable to read and write on the tape. A turning machine recognize input on the tape, manipulate and write the output on the same tape. A turning machine models all kind of computation, recognition, counting, computation, pattern matching and evaluation. Interestingly, any algorithm can be model with turning machine.

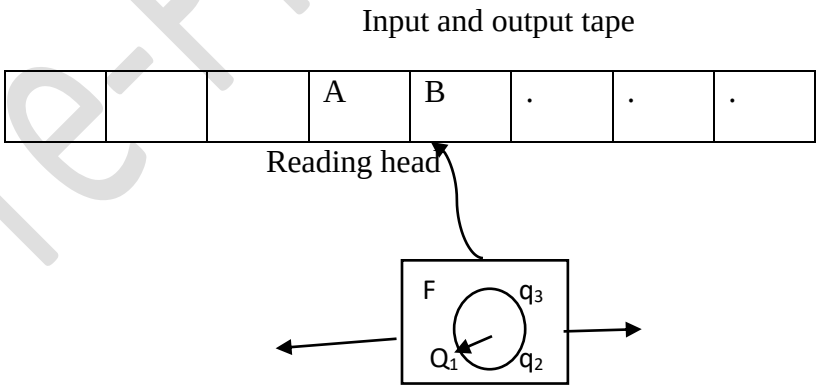


Figure 1.5: A model of Turing machine

3.6 Definition of Terms

3.6.1 Alphabet:

A finite, not empty set of symbols is called an alphabet. It is denoted by £ (sigma).

Examples:

£ = {0,1}, the binary alphabet
£ = {j,k}
£ = {a,b,c,…z} , the set of lowercase letters
£ = set of ASCII characters.

3.6.2 String:

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

Example: if £ = {a,b}. Then a, abab,aaabbb,aaaa,bbb are strings from £.

Example: given £ = {0,1}. Then 0, 01, 0001, 111001 are strings.

i. **Empty string**: a string with zero occurrences of symbols. It is also called null string. The string is denoted by £ (epsilon) or Λ (lambda) and may be chosen from any alphabet.
ii. **Length of a string**: the number of positions for symbols in the string. Example 0101100 has length 5. The length of string W is denoted as |W|.
Example: if W = abaabbb, Z = 1011
|W| = 7, |Z| = 4, |£| = 0.

3.6.3 Power of Alphabet

If £ is an alphabet, we can express the set of all strings of a certain length from that alphabet using the exponential notation.

$£^k$: the set of all strings of length k each of which is in £

$£^0$: {£}, regardless of what alphabet £ is. That is £ is the only string of length 0.

If £ = {0,1} then,

1. $£^1$ = {0,1}
2. $£^2$ = {00, 01, 10, 11}
3. $£^3$ = {000, 001, 010, 011, 100, 101, 110, 111}

3.6.4 Kleene star

£* is defined as the set of all strings over an alphabet £, including the empty string.

$\{0,1\}^* = \{£, 0, 1, 00, 01, 10, 11, 000, 001….\}$

$£^* = £^0 \cup £^1 \cup £^2 \cup …$

The symbol * is called Kleene star and is named after the mathematician and logician Stephen Cole Kleene.

£* is defined as the set of all string over an alphabet £ including the null string.

$£^* = £^1 \cup £^2 \cup £^2 …..$

Thus $£^* = £^* \cup \{£\}$.

3.6.5 Concatenation

Given two strings x, y ϵ £*, the concatenation of x and y is the string formed by placing a copy of y directly after a copy of x. it's denoted by x.y or xy

i.e. if given £ = {a, b}, x = aabba, y = abb

xy = aabbaabb, yx = abbaabba

Definition: for any string W, the equation

EW = WE = W hold.

i.e. £ is the identity for concatenation.

So formally, if S and T are subset of £*, then

S.T ={s,t| s ϵ S, t ϵ T}

3.6.6 Languages

A language defined over an alphabet £ is the set of strings with the same properties or characteristics all of which are chosen from £*.

Formally, if £ is an alphabet and L≤ £*, then L is a language over £.

Examples:

1. The language of strings of odd length define over £ = {a} is
   L = {a, aaa, aaaaa …} or precisely
   $L = \{a^n: n = 2n+1, £ = \{a\}\}$
2. L = {bw|cw: w ϵ £ *} define over £ = {a, b, c}

3. L = {w 0 1: w ϵ {0, 1}* }
4. L = {ww: w ϵ {a, b}*}
5. Language EQUAL defined over £ = {a,b}
   L = {w: w ϵ {a, b}* and $|w|_a = |w|_b$}. The language of strings with equal number of a's and b's
6. L = {a^n b^n, £ = {a, b} n = 1, 2, 3…}. The language of strings with a's followed by equal number of b's
7. Language PALINDROME: the language consisting of Λ and the string s defined over £ such that

   Rev(s) = s.

   Example; given £ = {0, 1}
   1. L = {ww^R: w ϵ £ *}. Is a palindrome
   2. L = {$w|w^R$: w ϵ £ *}. Is a palindrome
   3. L = {$w0w^R$: w ϵ £ *}. Is a palindrome
8. ϕ = an empty language
9. L = {£}. A language with one element which is empty string
   Note: ϕ ≠ {£}

3.6.7 Regular expression

A regular expression over £ is an expression that can be obtained from the basic alphabetical element using the operation of union, concatenation and Kleene. It is an explicit formula that describe a language defined over £.

Formally, a regular expression (over an alphabet £) is defined recursively as follows:

1. £ is a RE denoting the language {£}
2. If a ϵ £, then a is a RE denoting language {a}
3. If r and s are RE denoting L(r) and L(s), then r+s is a Res, denoting the language.{r}∪{s} or {r, s} (r) (s) is a RE denoting the language {r} {s} or {rs}
   r* is a RE denoting language. {r*}

Only the expression that can be obtained from the combination of I-IV are regular expression defined over £.

1. (a+b) (a+b) describes the set {aa, ab, ba, bb}
2. a* describes the set {£, a, aa, aaa}
3. (a+b)* denotes the set {£, a, b, ab, ba, aa, bb, aab….}. the set of all string of a's and b' including £.
   Note: (a+b)* = (a*b*)*

4. a+a*b generates {a,b,ab,aab,aaaab,aaaaab}

If we adopt precedence rule for operators, we apply closure, then concatenation, then alternation or union in order of precedence except changed by parenthesis.

Example:

1. a(a+b)* describes the set of a strings starting with a . i.e, {a, ab, aab, aba, abb….}.
2. Given £ = {0,1}. The R.E (0+1)* 01 (0+1)* represents the language of strings containing 01 as substring.
3. 0*1* represent a language with an arbitrary number of 0's followed by a random number of 1's precise'
   L = $\{0^n1^m; m,n >, 0\}$

Properties of R.E (Algebraic)

If r, s and t are RE,s

1. The operator union is commutative i.e.
   r+s = s+r (r|s =s|r)
2. The union operator is associative i.e.
   r+(s+t) = (r+s) + t
3. Concatenation is associative i.e.
   (rs) t =r(st)
4. Concatenation is distributive over union or alternative
   r(s+t) =rs +rt
   (s+t) r = sr + tr
5. £ is the identity for concatenation
   £r = r
   r£ = r
6. Kleene star is idempotent i.e.
   r** = r*
   Even ((r*)*)* = r*

4.0 Summary

In this unit you have learned:

- Both the formal and informal definition of automata theory and its associated concepts.
- How to represent an automaton using diagraph called state chart
- The categories and types of automata

5.0 Self-Assessment Questions

1. What is an automaton

2. Describe the main objective of automata theory
3. List and describe the types of automata in theoretical Computer Science

6.0 Tutor-Marked Assignment

1. Using examples of your own, give a detail description of the following:
   i. Finite Automaton
   ii. Language
   iii. Grammar
   iv. Regular expression

2. Describe simply as possible the language corresponding to each of the following RE
   i. $((0+1)^3)^* (0+1)$
   ii. $(1+10)^* (0+01)^*$
   iii. $(a+b)^* ab (a+b)^* + (a+b)^* ba (a+b)^*$

7.0 References and Further Reading

Martin, J.C.(2011). Introduction to Languages and The Theory of Computation (4th Edition). McGraw-Hill, New York, USA.

Mishra, K.L.P. & Chandrasekaran, N. (2008). Theory of Computer Science: Automata, Languages and Computation (3rd Edition). Prentice-Hall of India (PHI), New Delhi, India

Sipser, M. (2013) Introduction to the Theory of Computation (3rd Edition). Cengage Learning, Boston, USA.

https://www.neuraldump.net/2017/11/proof-of-kleenes-theorem/

http://www.cs.cornell.edu/courses/cs2800/2017sp/lectures/lec27-kleene.html

https://www.slideshare.net/samitachanchal/kleenes-theorem-48133245

https://www.tutorialspoint.com/automata_theory/deterministic_finite_automaton.htm

Module 2

Unit 1: Regular Expressions and Regular Languages

1.0 Introduction

In this unit, we will explore the fundamental concepts of regular expressions and regular languages, which are essential components of automata theory and formal language theory.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

i. Construct regular expressions that accurately represent specific sets of strings
ii. Identify and differentiate between regular languages and non-regular languages
iii. Apply regular expressions in practical scenarios, such as text processing and pattern matching, to solve problems related to string recognition and manipulation.

Definition: if r is a regular expression, then the language represented by r is denoted by L(r). Further, a language L is said to be regular if there is a regular expression r such that L = L(r).

A regular language over an alphabet £ is the one that can be obtained from the empty set (ϕ), {£}, and {a} for aϵ£, by finitely many applications of union, concatenation and Kleene star. Thus, a regular language is the set of all string that can be derived or generated from a regular expression.

Examples

i. The language L = $\{a^n | n \geq 0\}$ is regular as it can be represented by the R.E a*.
ii. The set of all string over {a,b} which contain ab as a substring is regular. For instance, the set can be written as
{xϵ{a,b}* | ab is a substring of x} = {yabz | y,z,ϵ{a,b}*} = {a,b}* {a,b} {a,b}*
Hence, the corresponding regular expression is (a+b)* ab (a+b)*
iii. The language L over {0,1} that contains 01 or 10 as substring is regular.
L = {x| 01 is a substring of x} $\cup$ {x| 10 is a substring of x}
={y01z|y,zϵ £*} $\cup$ {u10u|u, vϵ£*}
=£* {01y£* $\cup$ £*{10}£*}. Since £*, {01} and {10} are regular, then L is regular and the R.E representing L can be written as (0+1)* 01 (0+1)* + (0+1)*

Equivalence of RE's

Definition: two RE's r_1 and r_2 are said to be equivalence if both represent the same language. i.e. $L(r_1) = L(r_2)$ in which case, we write $r_1 \approx r_2$.

Prove:

Example:

i. The RE's (10+1)* and ((10)*1*)* are equivalent

Proof:

Since $L((10)^*) = \{(10)^n | n \geq 0\}$ and $L(1^*) = \{1^m 1^m | n \geq 0\}$ we have

$L((10)^*1^* = \{(10)^n 1^m 1^m, n \geq 0\}$

From above, we can imply that

$L((10)^*1^*)^*) = \{(10)^{n1} 1^{m2} (10)^{n2} 1^{m2} \ldots (10)^{n1} 1^{m1} 1^{n1} \geq 0$ and $0 \leq i \leq 1\}$

$= \{ x_1 x_2 \ldots x_k \mid x_i = 10 \text{ or } 1\}$ or $k = \sum_{i=0}^{1}(m1 + m2) \leq L((10 + 1)^*)$

Hence $L((10 + 1)^*) = L(((10)^*1^*)^*)$ and consequently $(10 + 1)^* \approx ((10)^*1^*)^*$

ii. Prove the formula $(111^*)^* = (11 + 111)^*$

Proof:

Let $L_1 = \{11\}$ and $L_2 = \{111\}$

Observe that $L_1 = \{11\} \leq L_1^* = \{11\}^*$ and $\{£\} \leq L_2^* = \{111\}^*$

Hence, we have by identity property

$L_1 = L_1 \{£\} \leq L_1^* L_2^*$

Similarly, $L_2 \leq L_1^* L_2^*$, hence $L_1 \cup L_2 \leq L_1^* L_2^*$

So $\{11\} \cup \{111\} = \{11\}^*\{111\}^*$

$= 11(\{£\}^* \{1\}^*)$

$= \{11\} (\{1\}^*) = \{11\} \{1\}^*$

Consequently,

$(\{11\} \cup \{111\})^* = (\{11\}\{1\}^*)^*$

And the corresponding RE is $(111^*)^*$ □

$(111^*)^* \approx (11 + 111)^*$

Let $r_1, r_2,$ and r_3 be any regular expression, the following properties hold for equivalence.

1. $r£ \approx £r \approx r$
2. $r_1 r_2 \approx r_2 r_1,$ in general
3. $r_1 (r_2 r_3) \approx (r_1 r_2) r_3$
4. $r\phi \approx \phi r \approx \phi$
5. $\phi^* \approx £$

6. $£^* \approx £$
7. If $£ \in L(r)$. then $r^* \approx r+$
8. $rr^* \approx r^*r \approx r+$
9. $(r_1 + r_2)\, r_3 \approx r_1\, r_3 + r_2\, r_3$
10. $r_1\, (r_2 + r_3) \approx r_1\, r_2 + r_1\, r_3$
11. $(r^*)^* \approx r^*$
12. $(r_1\, r_2)^*\, r_1 \approx r_1\, (r_2\, r_1)^*$
13. $(r_1 + r_2)^* \approx (r_1 * r_2 *)^*$

Exercise 2:

Prove that $b^+ (a^*b^* + \epsilon)\, b \approx b^+a^*b^*$

Unit 2 Finite State Machines

1.0 Introduction

In this unit, we will delve into the concept of finite state machines (FSMs), which are a fundamental model of computation used to represent and control the behavior of systems with a finite number of states.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

i. Analyze the behavior of finite state machines by tracing the processing of input strings and determining their acceptance or rejection based on the defined states.
ii. Compare and contrast the characteristics and capabilities of DFAs and NFAs
iii. Design deterministic and nondeterministic finite state machines for given languages

Finite Automaton

Finite Automaton is a computing machine that accept and recognize languages having finite representation. Thus, kind of machines are restricted systems with finite number of states only. In such system we have transitions between states on symbols of the alphabet. This we may refer them as finite state transition system. As the transitions are predefined in a finite state transition system, it automatically changes states based on the symbols given as input. Therefore a finite state transition system can also be allowed as a finite state automaton— a device that works automatically. The machine is of two types DFA &NFA.

Deterministic Finite Automata (DFA)

Deterministic finite automaton is a type of finite automaton in which the transitions are deterministic, in sense that there will be exactly one transition from a state on an input symbol.

Formally, a DFA is a quintuple (5-tuple) $D = (Q, £, q_0, \delta, F)$

Where

Q is a finite set of states

£ is a finite set of input alphabet

$q_0 \in Q$, a distinguished state called initial or start state

$F \subseteq Q$, the set of final or accepting states

δ is the transition function represented as

$\delta: Q \times £ \longrightarrow Q$

Note: that for every state and an input symbol, the transition function δ assigns a unique next state i.e. $\delta (q, s) \in Q \times £ \longrightarrow q^1 \in Q$

Example: given an automata $D = \{q_0, q, q_2\}, \{a, b\}, \delta, q_0, \{ q_2\}$

δ =

δ	a	b
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_2

Transition table:

The quintuple of a DFA can be simply represented with transition table instead of explicitly giving all the component. We can simply point out the initial and the final state of the DFA in the table of transition function (called transition table). We may use an arrow to point to the initial state and we encircle all the final states or append to the initial state and + to final states.

For example: the DFA above can be represented as

δ	a	b
→ q_0	q_1	q_0
q_1	q_2	q_0
(q_2)	q_2	q_2

Or

δ	a	B
q_0 --	q_1	q_0
q_1	q_2	q_0
q_2 +	q_2	q_2

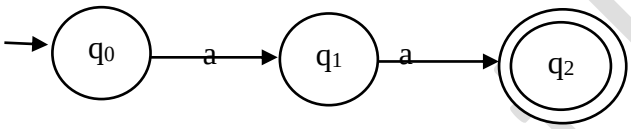
Transition Diagram

Normally, we associate some graphical representation to understand abstract concepts better. In the present context also we have a diagraph representation for DFA, (Q, £, q_0, δ, F), called a state transition diagram or simply a transition diagram.

The transition can be constructed as follows

1. Every state in Q is represented by a node
2. If δ(e,a) = q, then there is an arc from p to q
3. If there are multiple arcs from labelled $a_1,\ldots a_{k-1}, a_k$
4. There are many arrow with no source into the initial state
5. Final state are indicated by double circle

Example: the transition diagram for the DFA given above is



Note: there are two transition from q_2 itself on symbol a & b. As indicated in the point 3 above, these are indicated by a single arc from q_0 to q_2 labelled a, b.

Extended Transition Function

Recall that the transition function assigns a state for each statement input symbol. This naturally can be extended to all strings in £* i.e. assigning a state for each state and input string.

The extended transition function $\delta^*: Q \times £^* \rightarrow Q$ is defined recursively as follows

For all $q \in Q$, $x \in £^*$ and $a \in £$

$\delta^* (q, £) = q$ and

$\delta^* (q, xa) = \delta (\delta(q,x), a)$

Example: using the DFA Diagram defined above, find

$\delta^* (q_0, abba)$

Solution

$\delta^*(q_0, abba) = \delta(\delta^*(a, abb), a)$

$= \delta(\delta(\delta^*(a, ab), b), a)$

$= \delta(\delta(\delta(\delta^*(a, a), b), b), a)$

$= \delta(\delta(\delta(\delta(\delta^*(q, £), a), b), b), a)$

$= \delta(\delta(\delta(\delta(a, a), b), b), a)$

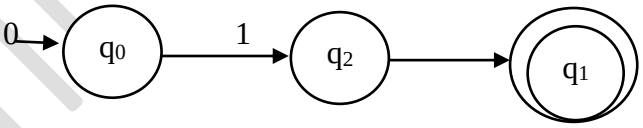
$= \delta(\delta(\delta(a, b), b), a)$

$= \delta(\delta(q_0, b), a)$

$= \delta(q_0, a) = q_1$

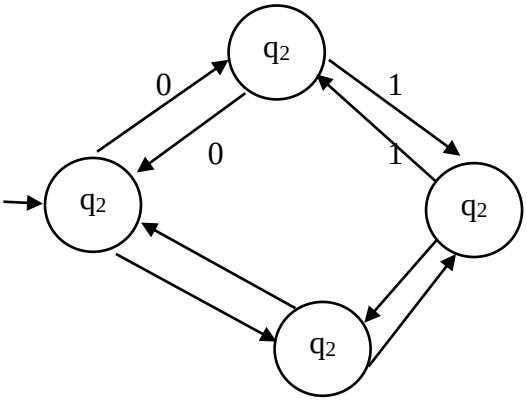
Example: a DFA D_1 that accept L = {a 01 y: x,y ϵ {0,1}*}

δ	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_2



Example: DFA D_2 accepting even number of 0's and even number of 1's

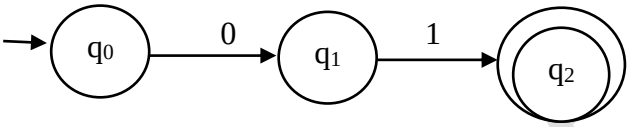
Even language



1 0

1 0

Example: DFA D3 accepting any string that ends with 01 defined on £ = {0,1}



Strings Accepted by DFA

A DFA accepts a string W = a_1, a_2, a_3….a_n if there is a path in the transition diagram that

1. Begins at a start state
2. Ends at an accepting state
3. Has a sequence of labels a_1, a_2, a_3….a_n

Example: considering DFA D3 above and strings 011101, 1001, 0101

1001 = q_0 1 q_0 0 q_1 0 q_1 1 q_2

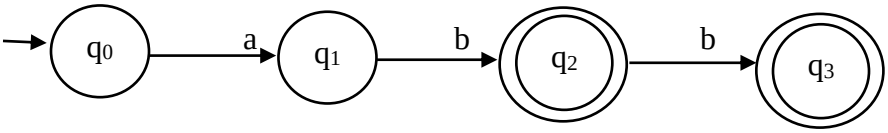
011101 = q_0 0 q_1 1 q_2 1 q_0 1 q_0 0 q_1 1 q_2*

0101 = q_0 0 q_1 1 q_2 0 q_1 1 q_2*

Formally, a string x ϵ £* is said to be accepted by a DFA

A = (Q, £, q_0, δ, F) if δ*(q_0, x) ϵ F

Use an example



Which of the following string can be accepted by DFA

i. aabbab
ii. abbabbab
iii. abbaabbb

Language of a DFA

The set of all strings accepted by the DFA D is said to be the language accepted by D and is denoted by L(D).

Formally,

L(D) = {x ϵ £* | δ*(q_0, x) ϵ F}

For example: considering the DFA A given above, what is the language of A

Solution

The only way to identify the language of the DFA is to list some strings accepted by the DFA starting from the simplest and find a pattern or uniform characteristic of the string.

Thus for A, the following strings can be accepted

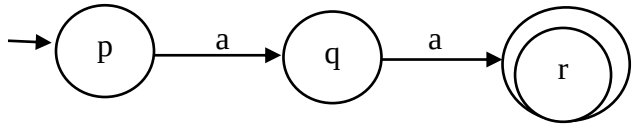
	ab	b	abb
	abb	b	ab
baa	ab	ab	ab
abb	abb	ab	abb
aaa	ab		

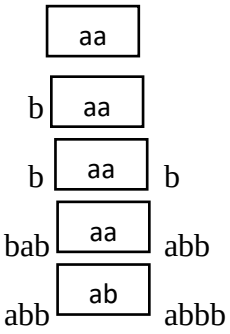
We can see that any of the strings either ends in ab or abb. So the language is given below

L(A) = { xab U xabb |x ϵ £*}

Example:

Given the DFA D below, what is the language of D





L(D) = {x aay| x,y ϵ {a,b}*}

Non- deterministic Finite Automaton (NFA)

This is an FA model that allows zero, one or more transitions from a state on the same input and / or transitions on no input.

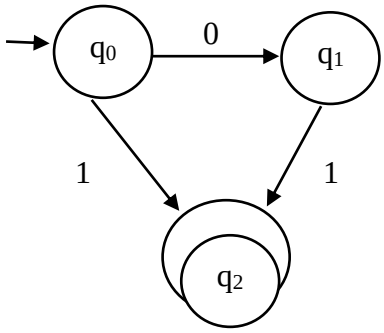
A NFA has the power to be in several states at once.

For instance, if one state has more than one transition for a given input symbol, then the automaton follows simultaneously all the transitions and get into several states at once. Thus, Non-determinism is recognized in FA if at least one of the following conditions:

i. there is at least one state with more than one transition on an input symbol a ϵ £

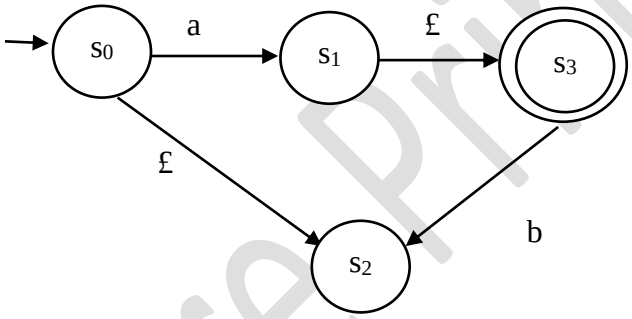
ii. there exist at least a transition with no input symbol called £-move or Λ-moves
iii. there exist a state with no transition on at least one input

For example, consider the FA given below



If the machine is in state q_0, and the input symbol is 0, from the figure above, it is clear that next state will be either (q_0), or q_1. Some moves of the machine cannot be determined uniquely by the input symbol and the present state such machine are called NFA.

Also, consider the FA below,



Note the transition s0 to s2 and s1 to s3 without any input i.e. £-transition. Let us consider the traces for the string ab from so

i. s_0 a s_1 b s_1 £ s_3*
ii. s_0 £ s_2 a s_2 b s_3*

The FA is a NFA

Formally, a NFA is a s-tuple (quintuple) (Q, £, ^δ, q_0, F)

Where:

Q: a finite set of states

£: a finite set of input symbols

^δ: *= ^δ (q, a) ϵ Q × £ → $\{q_i, q_j \ldots..\} \subseteq Q$: a transition function specifying for each state q and each input symbol s, the next state(s) $\{q_i, q_j \ldots..\}$ of the following automaton.

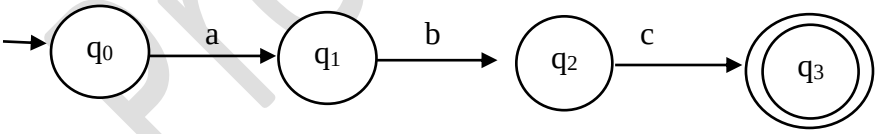
, $q_0 \epsilon$ Q: the initial state

F C Q: a set of accepting state

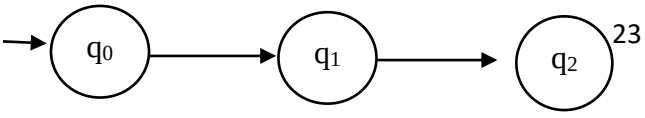
Example: given the following transition table

→ ^δ	a	b	C
q_0	$\{q_0 q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	ϕ	$\{q_2\}$	ϕ
q_2	ϕ	ϕ	$\{q_3\}$
q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$

The NFA is drawn below



Example: consider the NFA N_1 accepting all strings ending in 01 as DFA D_3.



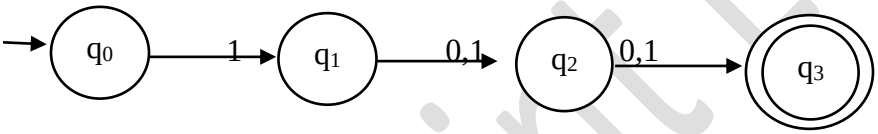
0 1

The example above illustrates how a NFA can be substantially simpler than equivalent DFA

Exercise: draw an NFA that accept language L = {w 01 w: w ϵ {0,1}*}

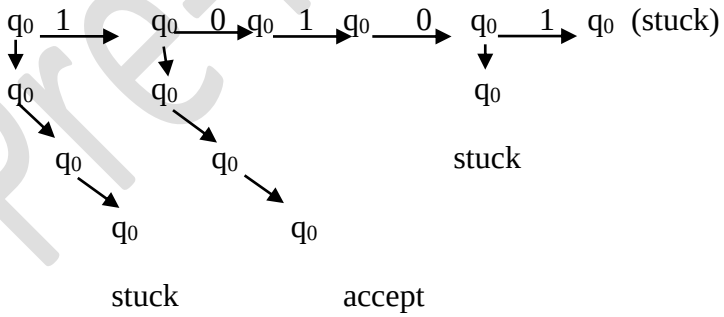
Note: NFA are usually simpler to implement and program in because they contain less number of states and less transition but less powerful than DFA as NFA can get stuck on the way if it follows wrong or otherwise backtrack.

Example: consider the NFA given below



Let us find δ* (q_0, 10101)

δ* (q_0, 10101) =



EQUIVALENCE of NFA and DFA

It is clear that each deterministic automaton is already a NFA i.e. the one that happens to always be just one state at once. Surprisingly, every language that can be described by a NFA can also be described by some DFA.

CASE 1: Multiple transition on an input

Theorem 1: for any NFA N, there exist a DFA D such that L(D) = L(N) and vice versa.

Proof: to prove the theorem, we construct a DFA D that can accept the same language as NFA N by subset construction.

This so called "subset construction proof" involves constructing all subset of the set of state of the NFA.

In short, since an NFA can only be in a finite number of state simultaneously, it can be seen as a DFA when each "super set" of the DFA corresponds to a set of states of the NFA.

Let N = {Q_n, £, ^δ_n, q0,F_n} be an NFA. We can construct a DFA D = {Q_o, £, δ_D, q_0, F_D} as follows.

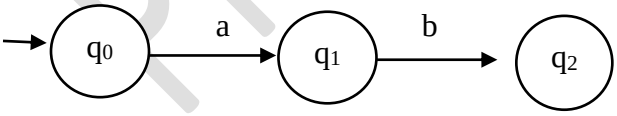
$Q_o = 2^{QN}$ that is v the power set of Q_o

Q_o (s, a) = Ʊ ^δ_n (p, a)

FD = {s C QN | snFN ≠ ϕ} i.e. FD is all sets of N's states that include at least one accepting states of Λ. It can easily be seen from the construction that both automata accept exactly the same input sequence.

Note: when N has n-states, the corresponding D has 2^n states. However not all states in D are reachable from {q_0} and those are likely to be garbage.

Example: convert the NFA given below to its equivalent DFA



δn =

	a	b
q_1	q_1 q_2	q_1
q_1	—	q_3
q_3	—	—

Step 1: construct the power set of $Q_n = \{q_1, q_2, q_3\}$

$=\{\phi, \{£\}, \{ q_1\}, \{ q_2\}, \{ q_3\}, \{ q_1 q_2\}, \{ q_1, q_3\}, \{ q_2, q_3\}, \{ q_1, q_2, q_3\}\}$

Step 2: construct a transition table δ for the subset using

$\Delta 0 (s, a) = \cup \wedge \delta_n (p, a)\ p \in s$

	a	b
ϕ	ϕ	ϕ
{q_1}	{ q_1,q_2}	{ q_1}
{ q_1}	ϕ	{ q_1}
{ q_1}	ϕ	ϕ
{q_1 q_2}	{ q_1,q_2}	{ q_1,q_3}
{q_1 q_3}	{ q_1,q_2}	{ q_1}
{ q_1}		{ q_3}
{ q_1,q_2,q_3}	{ q_1,q_2}	{ q_1,q_3}

Step 3: remove garbage unreachable states from δ_D.

Algorithm for removing garbage

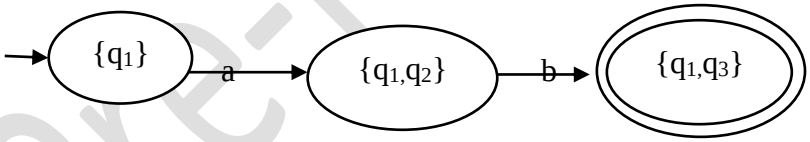
1. Mark the start state
2. For any marked state p, mark every state Q ɜ: there is a transition from P to Q on some input symbol
3. Repeat step 2 until no more states can be marked

After the whole process, the states remaining unmarked are garbage i.e. they are not reachable from the start state.

4. Draw the equivalent DFA

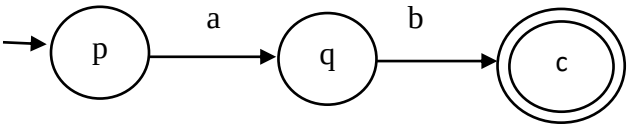
The new $\hat{\delta}_D$ after step3 is

	a	b
$\{q_1\}$	$\{q_1,q_2\}$	$\{q_1\}$
$\{q_1,q_2\}$	$\{q_1,q_2\}$	$\{q_1,q_3\}$
$\{q_1,q_3\}$	$\{q_1,q_2\}$	$\{q_1\}$



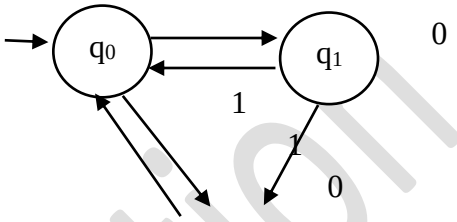
Note: all state in D containing at least a state in F_N are finite

For simplicity we can use letters to represent the subsets

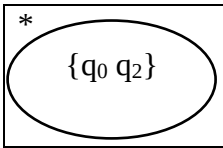
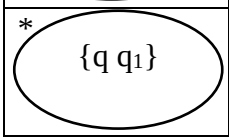


Example: convert the NFA given in table below to its equivalent DFA

	0	1
q_0	q_2	q_0 q_2
q_1	q_2	q_0
q_2*	q_0	—



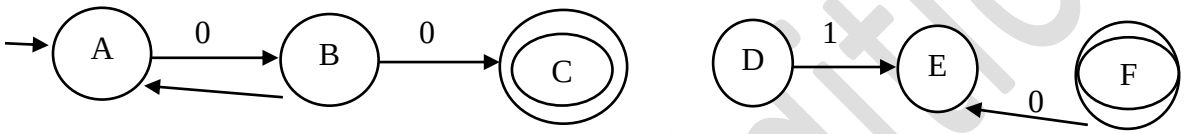
	0	1
ϕ	ϕ	ϕ
{q_0}	{ q_1}	{ q_0,q_2}
{q_1}	{ q_2}	{ q_0}
* {q_2}	{ q_0}	ϕ
{q_0 q_1}	{ q_1,q_2}	{ q_0,q_2}

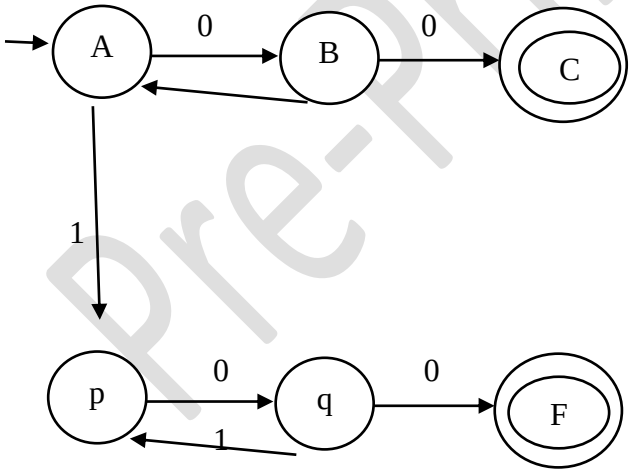
* 	{ q_0,q_1}	{ q_0,q_2}
* 	{ q_0,q_2}	{ q_0}
q_0,q_1,q_2	{ q_0,q_1,q_2}	{ q_0,q_2}

	a	b
{q_0}	{q_1}	{ q_0,q_2}
{q_1}	{q_2}	{q_0}
*{ q_2}	{q_0}	ϕ

{ q_0,q_1}	{ q_1,q_2}	{ q_0,q_2}
*{ q_0,q_2}	{ q_0,q_1}	{ q_0,q_2}
*{ q_1,q_2}	{ q_0,q_2}	{q_0}

D equivalent DFA is given below\





Case 2: NFA with £-moves

Recall that an NFA can also be represented using transitions with no input symbols or alphabet. This is called £-moves or £-transitions.

Theorem 2: if there is a NFA N with £=moves, then there exit an equivalent DFA D which has equal string recognizing power i.e. accepts same language as N. thus, if some NFA N with £-moves accepts string w, then there exists an equivalent DFA D that also accepts w.

Equivalent DFA can be constructed from NFA with £-moves using the £-closure method. To use this method, we define the following two operations.

1. £-closure (q): set of all states which are reachable from state q via or more £-transition alone. It is equivalent to one state of equivalent DFA.
2. Move (q, a): set of all states reachable from state q on input alphabet a

Using the above operations, the next state from state q on input symbol a is given as:

E (move (q, a))

Note £-closure (ϕ) = ϕ

Algorithm for converting £-NFA to DFA

Input: £-NFA N

Output: a DFA with states D states and transitions D trans

Let s be a state in N and T be a set of states

Find E (s_0)

Add state T = E (s_0) unmarked to D states

While ɔ unmarked state T in D states

Mark T for each input a

U = E (move (T, a))

If U is not ϵ D states then add U to D states unmarked D trans [T, a] = u

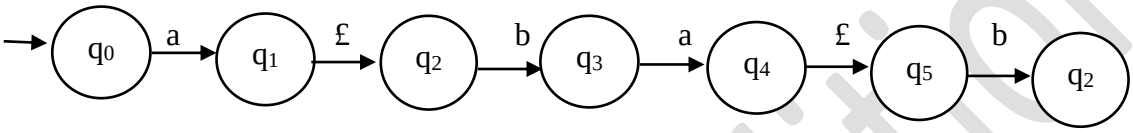
End for

End while

Note: E (s_0) is the initial state of D

A state D is accepting or final if it contains at least one accepting state in N.

Example: consider the following £-NFA. Construct an equivalent DFA

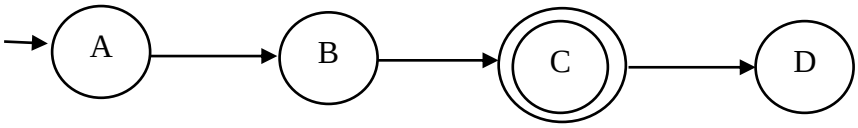


Using the algorithm

E (q_0) = {q_0}

	a	b
{ q_0}	{q_1,q_2,q_5}	ϕ
{q_1,q_2,q_5}	ϕ	{ q_6,q_3}*
*{ q_3,q_6}	{q_4,q_5,q_2}	ϕ
{q_4,q_5,q_2}	ϕ	{ q_6,q_3}

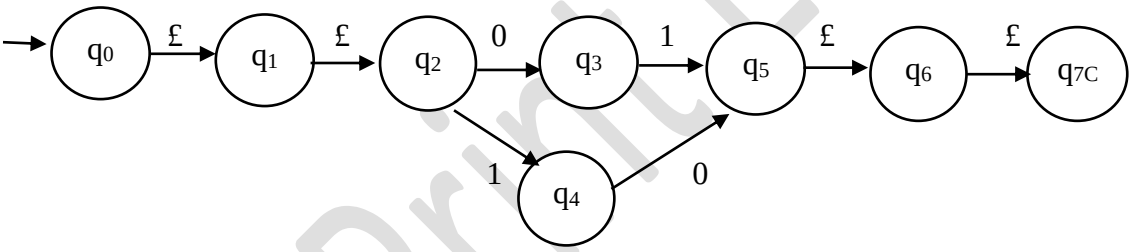
Equivalent DFA is given below



a b a

b

Example: construct an equivalent DFA for the £-NFA given below



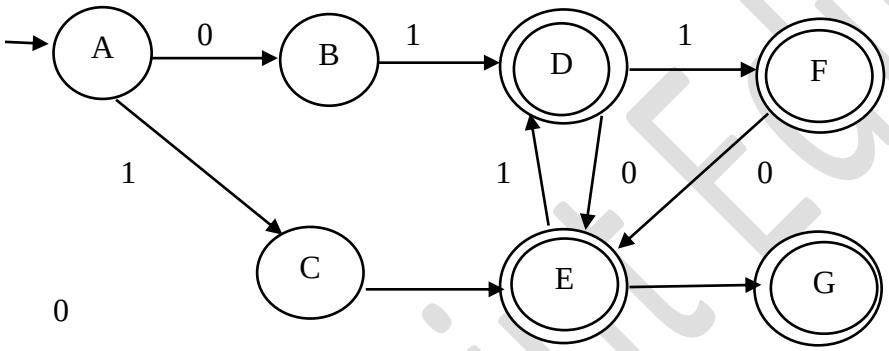
Solution

E (q_0) = {q_0, q, q_2}

	0	1
{q_0,q_1,q_2}	{q_1,q_2,q_3}	{q_1,q_2,q_4}
{q_1,q_2,q_3}	{q_1,q_2,q_3}	{q_n,q_2,q_4,q_5,q_3}
{q_1,q_2,q_4}	{q_1,q_2,q_3,q_5,q_7}	{q_1,q_2,q_4}
{q_1,q_4,q_5,q_6,q_7}*	{q_1,q_2,q_3,q_5,q_6,q_7}	{q_1,q_2,q_4,q_6,q_7}

$\{q_1,q_2,q_3,q_5,q_6,q_7\}^*$	$\{q_1,q_2,q_3,q_6,q_7\}$	$\{q_1,q_2,q_3,q_5,q_6,q_7\}$
$\{q_1,q_2,q_3,q_4,q_6,q_7\}^*$	$\{q_1,q_1,q_2,q_3,q_5,q_6,q_7\}$	$\{q_1,q_2,q_4,q_6,q_7\}$
$\{q_1,q_2,q_3,q_6,q_7\}^*$	$\{q_1,q_2,q_3,q_6,q_7\}$	$\{q_1,q_2,q_4,\ q_5,q_6,q_7\}$

The equivalent DFA is given as follows



Exercise

Find the equivalent DFA for the following NFA

Steps needed for proving a language is not regular using pumping lemma

Step 1: assume that L is regular and let n be the number of states in the corresponding FA

Step 2: choose a string x, such that W ≥ n, use pumping lemma to write x = uvw with $|uv| \leq n$ and $|v| > 0$

Step 3: find a suitable integer m such that $uv^m w$ is not ϵ L, this contradicts the assumption. Hence L is not regular.

Case 3

Combination of 1 and 2

It is possible to have a NFA with multiple transition on an input and at the same time contains £-transition. To convert the type of NFA to equivalent DFA, the following two lemma's are applied.

i. For every £-NFA N, there exist an equivalent NFA N without £-transition that accepts the same L as N.
ii. For every NFA N with no £-transition, there exist an equivalent DFA that accept the same L as N

*read about the proves of the above lemmas

Removal of £-transition from £-NFA

In order to convert an £-NFA to NFA, we use two operations

i. The £-closure (E(q))
ii. The extended transition function ^δ* which represent the set of next states for a state via string.

We define the extended transition function

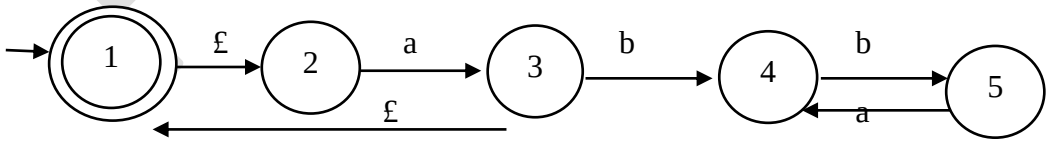
$\hat{\delta}: Q \times £^* \longrightarrow p(Q)$

As follows

1. $\hat{\delta}(q, £) = E(q)$ ∀ $q \in Q$
2. $\hat{\delta}(q, xa) = E(U\ \delta(p, a))$ ∀ $q \in Q$, $y \in £^*$ and $q \in £$

Where E (q) is the £-closure of q and x is a string

For example: given the £-NFA below and its transition



q	δ (q,a)	δ (q,b)	δ (q,£)

1	ϕ	ϕ	{2}
2	{2, 3}	ϕ	ϕ
3	ϕ	{4}	ϕ
4	ϕ	{5}	{1}
5	{4}	ϕ	ϕ

In other to eliminate £-transition and obtain an equivalent and obtain an equivalent NFA, we need to eliminate ^δ (q, a) for all states q and for all input symbols

To find the ^δ (q, a) we take the following steps

i. Find ^δ (q, a) using E (q)
ii. Find the ^δ (q, a) using the formula

^δ (q, a) = E (U δ (p, a)) p ϵ ^δ (q, £)

In using the state 1 in the £-NFA above we can compute ^δ (1, a) as follows

^δ (1, £) = E (1) = {1, 2}

^δ (1, £a) = E (U {δ (k, a) | k ϵ δ* (1, £)})

= E (δ (1, a) U δ* (2, a))

= E (ϕ U (2, 3))

= E ({2, 3})

{2, 3}

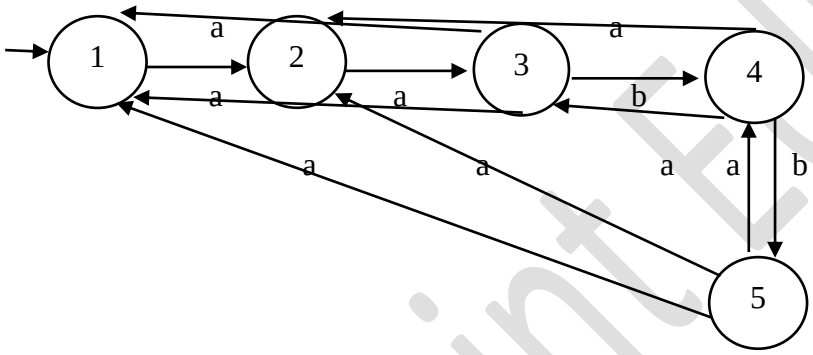
^δ (1, a) = {2, 3}

So, we apply the same step for all states on all inputs. The resulting extended transition table is given below

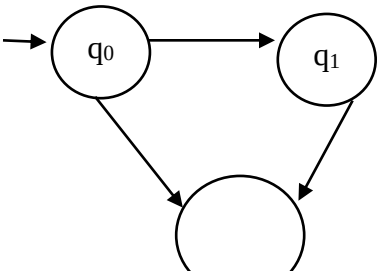
q	^δ (q,a)	^δ (q,b)

1	{2, 3}	ϕ
2	{2, 3}	ϕ
3	ϕ	{1,2,4}
4	{2,3}	{5}
5	{1,2,4}	ϕ

The NFA is given below

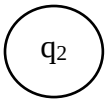


Example consider the following £-NFA, construct an £-transition free NFA equivalent



£,b

£ b



Solution

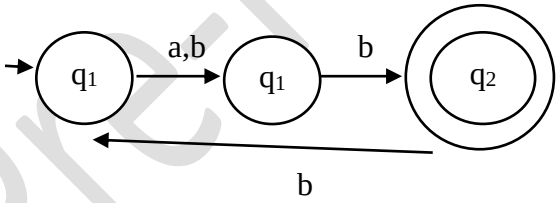
The transition table

q	δ(q,a)	δ (q,b)	δ (q,£)
q_0	ϕ	$\{q_0,q_2\}$	$\{q_1,q_2\}$
q_1	$\{q_1\}$	$\{q_1,q_2\}$	ϕ
q_2	ϕ	ϕ	ϕ

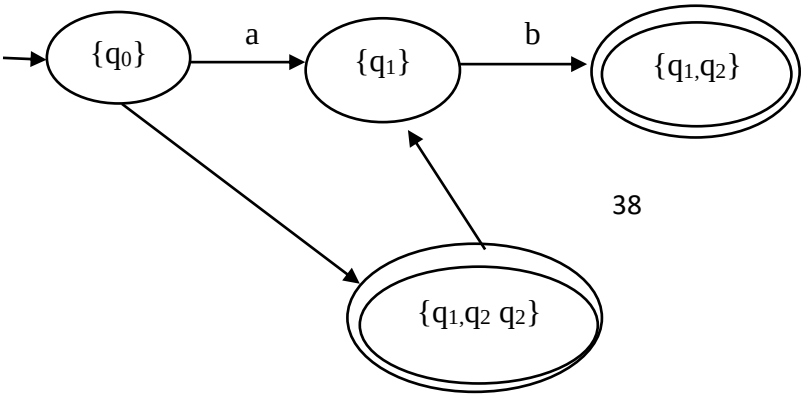
extended transition table

q	^δ(q,a)	^δ(q,b)
q_0	$\{q_1\}$	$\{q_0,q_2,q_1\}$
q_1	$\{q_1\}$	$\{q_1,q_2\}$
q_2*	ϕ	ϕ

The NFA is given below



The DFA is given below



a a

From here we use subset on structure method to convert the NFA to DFA

	a	a
$\{q_0\}$	$\{ q_1\}$	$\{ q_0, q_1,q_2\}$
$\{q_1\}$	$\{ q_1\}$	$\{ q_1,q_2\}$
$\{q_2\}$	ϕ	ϕ
$\{q_0\ q_1\}$	$\{q_1\}$	$\{ q_0, q_1,q_2\}$
$\{q_0\ q_2\}$	$\{q_1\}$	$\{ q_0, q_1,q_2\}$
$\{q_1,q_2\}$	$\{q_1\}$	$\{ q_1,q_2\}$
$\{q_0,q_1,q_2\}$	$\{q_1\}$	$\{ q_0, q_1,q_2\}$

Module 3 Regular Languages and Finite Automata

Unit 1 Kleen's Theorem
Unit 2 Pumping Lemma for Regular Languages
Unit 3 Distinguishable Strings

Unit 1: Kleen's Theorem

1.0 Introduction

This unit exposes you to the equivalence of Regular Expression (RE) and Finite Automaton (FA) based on Kleen's theorem. You will be able to learn kleen's theorem and how to use it to construct FA's. The principles or rules for building FAs from RE are explained and practical examples are presented.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

iv. State the Kleen's theorem for regular languages
v. State the principles of building FA's using union, concatenation and closure
vi. Construct an FA from a given Reusing the Kleen's theorem

3.0 Main Content

3.1 Kleen's Theorem (10Mins)[SAQ1]

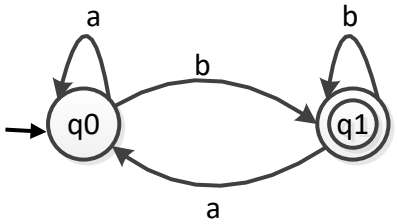
Recall that Regular Languages (RL) are class of languages generated from some regular expressions and the class of languages recognized by FA are regular languages. Thus, there is a direct equivalence between regular languages and FA (DFA & NFA), this equivalence is established by **Kleene's theorem**.

The theorem states that ***"if a language L can be expressed by a RE (i.e. if a language is regular), then there exist an FA that can accept L and vise-versa".***

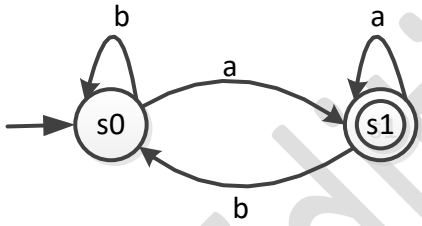
Proof: As REs are obtained by applying union, concatenation and closure on the elements of £ and ε, it is considerably sufficient as a proof, if we can show that FAs can be built using the same operations.

[SAQ2,3]

1. **Union of two FAs**
   Given that regular expression $r_1 = (a+b)^*b$ defines language L_1, the FA_1 is



Given also that $r_2 = (a+b)^*a$ defines L_2 and FA_2 be



Let FA_3 be an FA corresponding to $r_1 + r_2$, then

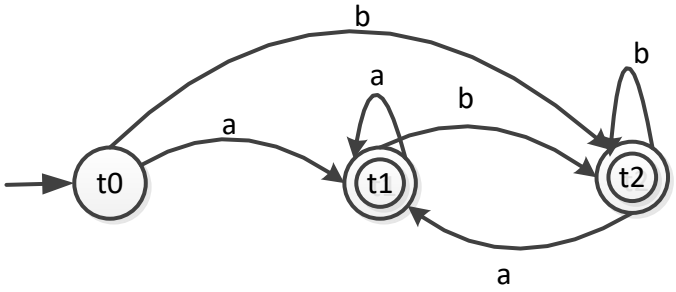
i. The initial state if FA_3 must correspond to the initial state of FA_1 or the initial state of FA_2
ii. A final state must correspond to a final state of FA_1 or FA_2 or both since L_3 corresponding $r_1 + r_2$ is the union of L_1 & L_2 consisting of strings belonging to L_1 or L_2 or both

To build the transition table of FA_3, we start from the initial state which is the union of the initial state of FA_1 or FA_2 and continue the transitions of new state discovered. All states containing final state in FA_1 or FA_2 or both are final states in FA_3.

The transition is given below:

States	**Transition**	
	A	b
$t_0 = (q_0, s_0)$	$(q_0, s_1) = t_1$	$(q_1, s_0) = t_2$
$t_1^* = (q_0, s_1)$	$(q_0, s_1) = t_1$	$(q_1, s_0) = t_2$
$t_2^* = (q_1, s_0)$	$(q_0, s_1) = t_1$	$(q_1, s_0) = t_2$

Thus, the corresponding FA_3 is given below



2. Concatenation of two FAs

Using the same FAs (FA_1 & FA_2) corresponding to r_1 & r_2, an FA_3 can be built corresponding to r_1 r_2 using the following principles

i. The initial state of FA_3 must correspond to the initial state of FA_1

ii. The finial state of FA_3 must correspond to the final state of FA_2 , so any state in FA_3 containing the final state in FA_2 is a final state.

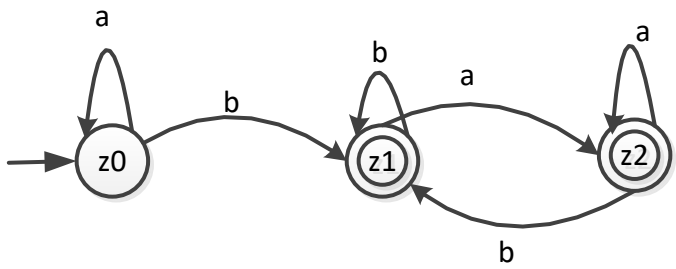
iii. Since the language corresponding to r_1r_2 is the concatenation of corresponding languages L_1 & L_2 which consists of strings obtained by concatenating strings of L_1 to those of L_2, therefore the moment the final state of FA_1 is reached, the possibility of the initial state of FA_2 will be added as well.

Using the principle above, the following transition table can be built for FA_3 corresponding to r_1 r_2.

States Transition

	a	b
$z_0 = q_0$	$q_0 = z_0$	$(q_1, s_0) = z_1$
$z_1 = (q_1, s_0)$	$(q_0, s_1) = z_2$	$(q_1, s_0) = z_1$
$z_2^* = (q_0, s_1)$	$(q_0, s_1) = z_2$	$(q_1, s_0) = z_1$

The corresponding FA_3 is given below:



3. Closure of an FA

Given a RE r_1 corresponding to FA_1, we can build a FA_1^* corresponding to r_1^* using the principle:

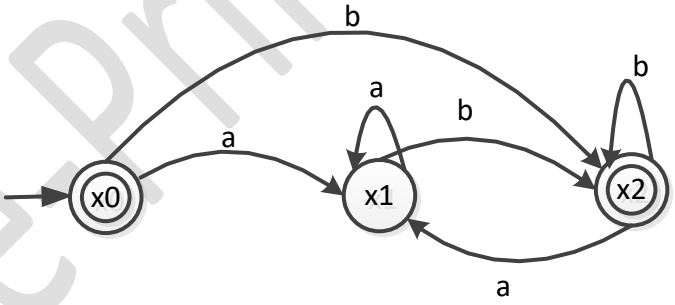
i. Closure of a FA_1 is the same as concatenation of FA_1 with itself, with the initial state of the required FA^* being the final state as well.

ii. If at the initial state of FA_1 there is either a loop or an incoming transition edge, the initial state corresponds to a final state and a non-final state as well.

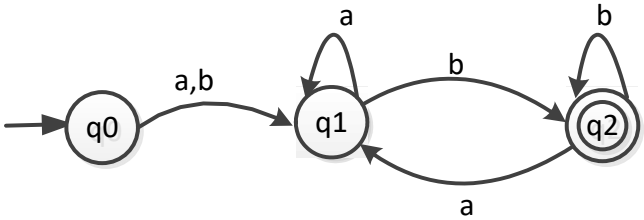
Using the $r_1 = (a+b)^*b$, we can build the FA_1^* that corresponds to $((a + b)^*b)^*$ based on the principle above to generate the transition table below:

δ	A	b
(final)$x_0^* = q_0$	$q_0 = x_1$	$(q_1,q_0) = x_2$
(non-final)$x_1 = (q_0)$	$q_0 = x_1$	$(q_1,q_0) = x_2$
$x_1^* = (q_1,q_0)$	$q_0 = x_1$	$(q_1,q_0) = x_2$

The corresponding FA_1^* is given below

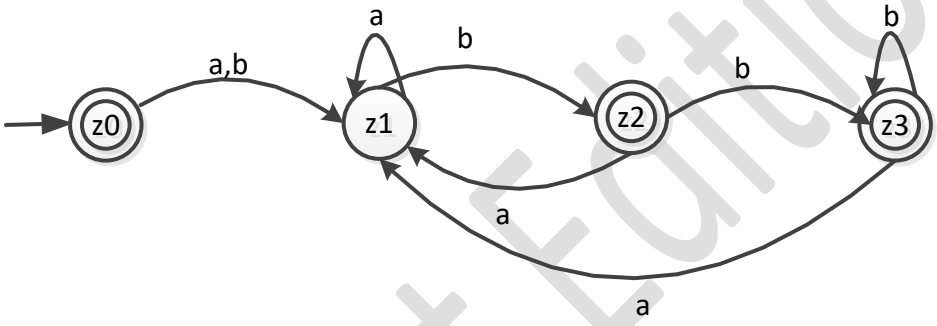


Consider an FA given below that corresponds to a RE r = (a+b) (a+b)*b with the initial state having no loop nor incoming edge.



Using the closure principle above, the transition table for FA* (closure of the FA) is given below followed by the transition diagram.

δ	A	b
(final)$z_0 = q_0$*	$q_1 = z_1$	$q_1 = z_1$
$z_1 = q_1$	$q_1 = z_1$	$(q_2, q_0) = z_2$
$z_2 = (q_2, q_0)$*	$q_1 = z_1$	$(q_2, q_0, q_1) = z_3$
$z_3 = (q_2, q_0, q_1)$*	$q_1 = z_1$	$(q_2, q_1, q_0) = z_3$



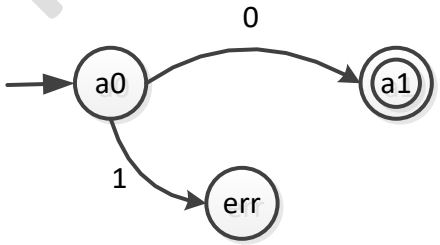
By applying the Kleene's theorem, FA accepting the same language as a given RE can be built. Simple FAs are built from each term of the RE and combined using concatenation, union and closure method.

Example: given that a language L can be described by the RE 0*1(0 +1)*, build an FA that can recognize L

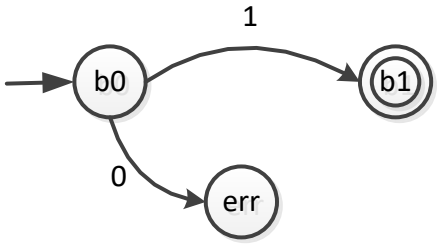
Solution:

Step 1: build FA for each term

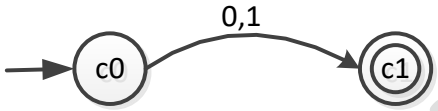
FA(0) =



FA(1) =



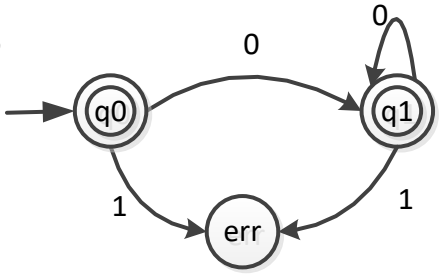
FA(0+1) =



Note: the err state in the FAs is error state in which when entered, the machine stops reading and reject the string being read.

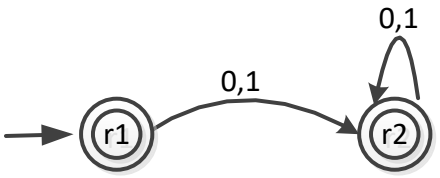
Step 2: Build FA (0*) following the kleen's closure principles.
The transition table and diagram for FA(0*) are given below:

	0	1
a_0* =q_0*	(a_0,a_1)* = q_1	err
(a_0,a_1)*= q_1	(a_0,a_1)= q_1	err



Step 3: Build FA(0+1)* using the closure principle as well.
The transition table and diagram for FA(0+1)* are given below:

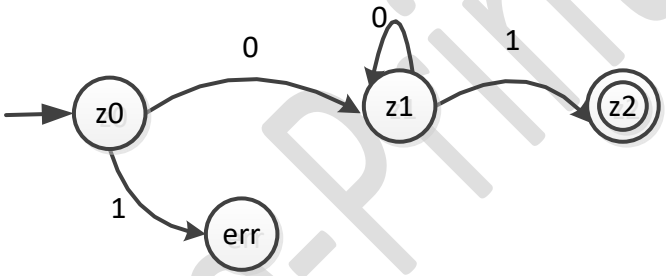
	0	1
r_0 =c_0*	(c_0,c_1)* =r_1	(c_0,c_1) =r_1
$_{r1}$ =(c_0,c_1)*	(c_0,c_1)= r_1	(c_0,c_1)= r_1



Step 4: Build FA(0*1) using concatenation principles.

The transition table and diagram for FA(0*1) are given below:

	0	1
z_0 =q0	(q1,b0)=z1	err
z_1 = (q1,b0)	(q1,b0)=z1	b_1 = z2
z_2 =(b_1)*	-	-

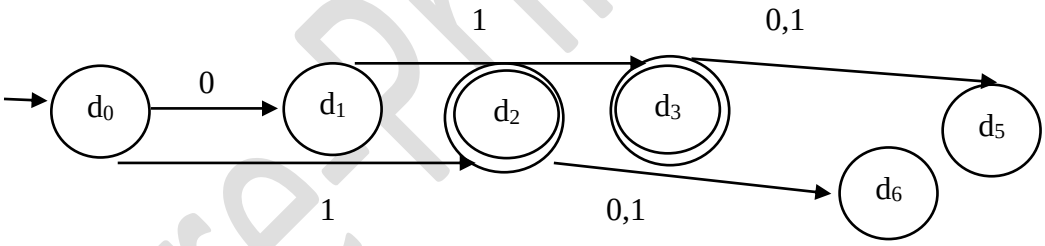


Step 5: lastly, Build FA(0*1 (0+1)*) using concatenation principle (concatenation)

	0	1
D_0 =z_0	z_1	(z_5,y_0)
D_1 = z_0	z_1	(z_3,y_0)

$D_2 = (z_2, y_0)^*$	(z_4, y_1)	(z_4, y_1)
$D_3 = (z_3, y_0)^*$	(z_5, y_1)	(z_5, y_1)
$D_4 = (z_4, y_1)^*$	(z_4, y_1)	(z_4, y_1)
$D_5 = (z_4, y_1)^*$	(z_5, y_1)	(z_5, y_1)

The FA (0*1 (0+1)*) is given below



The Kleene's theorem can also be used to prove that a language is regular. It follows that "a language L over the alphabet £ is a regular if and only if there exists a FA with input alphabet £ that accepts L".

For us to do that, the following steps must be taken

i. Describe the given language L by a R.E
ii. Use Kleene's methods to build the FA from the R.E

4.0 Summary

In this unit you have learned that:

- Kleene's theorem states that "if a language L can be expressed by a RE, then there exist an FA that can accept L and vise-versa".
- It is considerably sufficient as a proof, to show that FAs can be built using the regular expression operations: union, concatenation and Kleen' closure,
- From a Given RE, Simple FAs are built from each term of the RE and combined using concatenation, union and closure method.
- By applying the Kleene's theorem, FA accepting the same language as a given RE can be built.
- The Kleene's theorem can be used to prove that a language is regular :"a language L over the alphabet £ is regular if and only if there exists a FA with input alphabet £ that accepts L".

5.0 Self-Assessment Questions

4. State Kleen's theorem
5. Highlight the principles of building FA's from union, concatenation and closure
6. Given the RE = (a|b)*ab, Construct an FA that accepts the same language as the RE

6.0 Tutor-Marked Assignment

3. Given a language **L = { bw | cw: w є □ *}** defined over an alphabet □ **= {a, b, c}**, build an FA that accepts L based on Kleen's theorem.

7.0 References and Further Reading

Martin, J.C.(2011). Introduction to Languages and The Theory of Computation (4th Edition). McGraw-Hill, New York, USA.

Mishra, K.L.P. & Chandrasekaran, N. (2008). Theory of Computer Science: Automata, Languages and Computation (3rd Edition). Prentice-Hall of India (PHI), New Delhi, India

Sipser, M. (2013) Introduction to the Theory of Computation (3rd Edition). Cengage Learning, Boston, USA.

https://www.neuraldump.net/2017/11/proof-of-kleenes-theorem/

http://www.cs.cornell.edu/courses/cs2800/2017sp/lectures/lec27-kleene.html

https://www.slideshare.net/samitachanchal/kleenes-theorem-48133245

Unit 2 Pumping Lemma for Regular Languages

1.0 Introduction

This unit exposes you to the concept of Pumping lemma for regular languages with focus on the Pumping property of regular languages adopted from finite automata, the use of pumping lemma to prove non-regularity of languages. It supplies you with practice examples and proof solutions.

2.0 Learning Outcomes

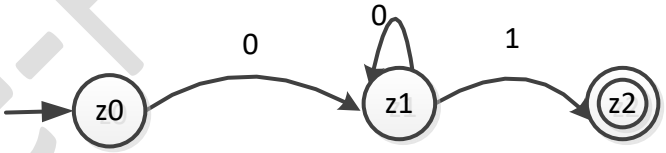
At the end of this unit, you should be able to:

i. Identify the pumping property of regular languages
ii. State the pumping lemma for regular languages
iii. Use pumping lemma to prove that a given language is not regular

3.0 Main Content

3.1 Pumping Property of Regular Languages (8Mins)[SAQ1]

Recall that every regular language can be accepted by a finite automation (FA), a recognizing device with a finite set of state and no auxiliary memory. We can use the finiteness of this machine to device another property shared by all regular languages. The property is that all strings in the language can be **"pumped"** if they are at least as long as a certain special value, called the **pumping length** to obtain some other strings which are member of the language. That means such string contains a section that can be repeated any number of times with the resulting string remaining in the language. For example, consider the FA below that accepts the language **L = {w1: w ϵ {0}*)** defined over □ **= {0,1}**:



You will discover that there is a loop at the state z1 over the string 0, so to get longer string member of the language, the string 0 can be pumped any number of times to generate other member of the language. Thus, if 0 is pumped 4 times, we have the string 000001, a member of the language. this is the property of FAs shared by all infinite regular languages.
However, **if we can show that a language does not have this property, we are guaranteed that it is not regular**. The technique for proving nonregularity of any language stems from a theorem about regular languages, traditionally called the **pumping lemma.**

3.2 The Pumping Lemma[SAQ2,3]

LEMMA:

Suppose L is a regular language recognized by a finite automation with n states. For any $x \epsilon L$ with $|x| \geq n$. String x may be divided into three pieces, $\boldsymbol{x = uvw}$ for some strings u, v and w satisfying the following conditions:

1. $\boldsymbol{|uv| \leq n}$
2. $\boldsymbol{|v| > 0}$
3. *The pumping lemma asserts that for any* $\boldsymbol{m \geq 0,\ uv^m w \epsilon L}$

We can think of it as saying that for an arbitrary string L, provided that it is sufficiently long, a portion of it can be "pumped up", introducing additional copies of the substring *v*, so as to obtain many distinct elements of L. From the lemma, when *x* is divided into *uvw* either *u* or *w* may be ε, but condition 2 says that $v \neq \varepsilon$. Observe that without condition 2 the theorem would be trivially true. Condition 1 states that *u* and *v* together have length at most *n*. It is an extra technical condition that we occasionally find useful when proving certain languages to be nonregular.

Showing that a language does not have this property will then be a way of showing that the language is not regular. In other to prove that a language L is not regular, we must show that L fails to have the property described in the lemma. We do this by assuming that the property is satisfied and then deriving a contradiction (proof by contradiction).

Example1: Given the language palindrome Lp =$\{ 0^n10^n \}$defined over £ = {0,1}, Show that Lp is not regular.

Assume that Lp is regular,
Let n be the number of states in a FA accepting Lp.
Choose any x in L satisfying $\boldsymbol{|x| \geq n}$
Let $\boldsymbol{x = 0^n10^n = uvw}$
Since $\boldsymbol{|uv| \leq n}$ *and* $\boldsymbol{|v| > 0}$ *by Pumping Lemma*
It means uv is in the first part of x, $\boldsymbol{0^n}$
Let $\boldsymbol{v = 0^j}$ *and* $\boldsymbol{uv = 0^n}$*, then* $\boldsymbol{u = 0^{n-j}}$
Now, $\boldsymbol{uvw = 0^{n-j}\,0^j10n}$
For any $\boldsymbol{m \geq 1}$*, we want to check if* $\boldsymbol{uv^m w \epsilon Lp}$ *or not.*
So, Let us test for $\boldsymbol{m \geq 2}$**. Let m = 2,**

$$\boldsymbol{uv^m w = 0^{n-j}(0^j)^2 10^n}$$
$$\boldsymbol{= 0^{n-j}\,0^{2j}10^n}$$
$$\boldsymbol{= 0^{n+j}10^n \text{ is not } \epsilon Lp}$$

Therefore, Lp is NOT regular.

Note: pumping lemma cannot sufficiently prove that a language L is regular (if L passes the pumping lemma test) , it can only be used to prove irregularity in languages.

Example 2: prove that the language L defined over £ = {a, b} is not regular

L = {x ϵ a* b* : $|x|_a = |x|_b$ }

Solution :
Suppose L is regular
Let ***n*** *be the number of states in a FA accepting* ***L***
We choose string $y = a^n b^n$, $|y| \geq n$
We break y into uvw satisfying $uv \leq n$, $v > 0$.
Thus, uv is contained in $a^n b^n$.
Let $uv = a^{n-1}$, *then*
$w = ab^n$
Assume $v = a^i$
$u = a^{(n-1)-i}$
The string y (uvw) becomes $a^{n-1-i}a^i ab^n$
Let $m = 1$
$= a^{n-1-i}(a^i)^1 ab^n$
$= a^{n-1-i}a^i ab^n$
$= a^n b^n$
Let m = 2
$= a^{n-1-i}(a^i)^2 ab^n$
$= a^{n-1-i}a^{2i} ab^n$
$= a^{n-1-i+2i} ab^n$
$= a^{n+i-1} ab^n$
$= a^{n+i} b^n$ ***not ϵ L*** *(This is a contradiction)*

Then L is not regular !!

3.2 Pumping Down

Sometimes **"pumping down"** is useful when we apply the pumping lemma on some languages.

Example 3: Use the pumping lemma to show that $\mathbf{L = \{0^i 1^j \mid i > j\}}$ is not regular.

To show that L is not regular, the proof is by contradiction with pumping down.

Solution:
Let's assume that L is regular.
Let n be the pumping length for L given by the pumping lemma.
Let $\mathbf{y = 0^{n+1}1^n}$.
Then y can be split into uvw, satisfying the conditions of the pumping lemma.
By condition $|uv| \leq n$, v consists only of 0s.
Let's examine the string uvvw to see whether it can be in L.
Adding an extra copy of v increases the number of 0s.
But L contains all strings in 0^*1^* that have more 0s than 1s.

So, increasing the number of 0s will still give a string in L (No contradiction occurs, even as we can see that the language is not regular).
We need to try something else.
The pumping lemma states that $uv^m w \in L$ even when **m = 0,**
so let's consider the string $uv^0w = uw$.
Removing string y decreases the number of 0s in y and recall that our y has just one more 0 than 1.
Therefore, **uw** cannot have more 0s than 1s.
So, it cannot be a member of L.
Thus, we obtain a contradiction.

Note: When applying the pumping lemma, what you are looking for intuitively is a contradiction, if natural values for m could not produce the contradiction, you should consider pumping down (make m=0).

4.0 Summary

So far in this unit, I had explained that:

- All FA has a pumping property that all strings in the language they accept can be **pumped** to obtain some other strings which are member of the language, if they are at least as long as a certain special value, called the **pumping length**.
- If we can show that a language does not have the pumping property, then the language is not regular.
- The pumbing lemma states that suppose a language L is regular and recognized by a finite automation with n states. For any x ϵ L with $|x| \geq n$ and some strings u,v, and w, string x may be split as **x = uvw** such that **$|uv| \leq n$ and $|v| > 0$.** Then for any **$m \geq 0$**, **$uv^m w \epsilon L$.**
- Pumping lemma cannot completely prove the regularity of a language, it can only prove that a language is not regular.

5.0 Self-Assessment Questions

i. Describe the pumping property of regular languages
ii. State the pumping lemma for regular languages
iii. Use the pumping lemma to prove that the language "palindrome" is not regular

6.0 Tutor Marked Assessments

1. Prove that the following Languages are not regular using the pumping lemma:
   L = {xay | x,y ϵ {a, b}*, |x| = |y|}
   L = {ww | w ϵ {a, b}*}

7.0 References and Further Reading

John, E.H, Rajheev, M. & Jeffrey, D.U. (2001). Introduction to Automata Theory, Languages and Computation (2nd Edition). Addison-Wesley, Pearson Education, USA.

Anderson, J.A. (2012). Modern Applications of Automata Theory (1st Edition). Cambridge University Press, London.

Sipser, M. (2012) Introduction to the Theory of Computation (3rd Edition). Cengage Learning, Boston, USA.

Tourlakis, G. (2012). Theory of Computation (1st Edition). Wiley, USA

https://www.geeksforgeeks.org/pumping-lemma-in-theory-of-computation/
https://www.tutorialspoint.com/automata_theory/pumping_lemma_for_regular_grammar.htm
https://www.sanfoundry.com/automata-theory-pumping-lemma-regular-languages/

Unit 3: Distinguishable Strings

1.0 Introduction

This unit discusses distinguishable strings as used in Regular Languages. It also explains the Myhill-Nerode Theorem built upon the distinguishable strings to build FAs for Regular Languages. Examples on how to use the theorem to prove the regularity of a language are also given.

2.0 Learning Outcomes

After you have studied this unit, you should be able to:

i. Define distinguishable string
ii. State Myhill-Nerode Theorem
iii. Build an FA from a given Language using the Myhill-Nerode Theorem

3.0 Main Content

3.1 Distinguishable Strings with Respect to a Language(10Mins) [SAQ1]

Let **L** be a language in **£*** and ***x*** be any string in **£***, the set **L/*x*** is defined as follows:

$$L/x = \{ z \in £^* \mid xz \in L \}$$

Definition: *Two strings x and y are said to be* ***distinguishable*** *with respect to a language* ***L****, if* $L/x \neq L/y$

Any string ***z*** that is in one of the two sets but not in the other i.e.

for which: $xz \in L$ and yz ***is not*** $\in L$ **or vice versa**

is said to distinguish ***x*** and **y** with respect to **L**.

If $L/x = L/y$, then ***x*** and **y** are said to be **indistinguishable.**

In order to show that two strings ***u*** and ***v*** are distinguishable with respect to language L, it is sufficient to find **one (1) string z** such that $uz \in L$ and vz ***not*** $\in L$ or **vice versa**.

Example1:
Consider the language:

$$L = (\{0,1\}\{0,1\})^*$$

The strings **x = 0** and **y = 01** are **disguisable over L**.
Prove:
Selecting a string in {0,1}* for z e.g. let **z = 10**
***x* = 0**, **y = 01**

*x*z = 010 not ϵ L
*x*z = 0110 ϵ L
let **z = 00**
*x*z = 000 not ϵ L
*x*y = 0100 ϵ L

In fact, any string in {0,1}* distinguishes both x, y.

Example 2:
consider the language L = {0,1}*{01}

The strings 1 and 10 are distinguishable with respect to L by only one string 1.

11 not ϵ L and 101 ϵ L

However, they are indistinguishable by any other string from {0,1}*.

let z = 01,

101 ϵ L and 1001 ϵ L

3.2 Myhill-Nerode Theorem [SAQ2,3]

Myhill-Nerode Theorem states that:

*Suppose **L** ⊂ **£*** and for some positive integer **n**, there are **n** classes of strings in **£***, any two of which are distinguishable with respect to **L**, then every **FA** recognizing **L** must be composed of at least **n** states.*

Since FAs accepts only regular languages, the above theorem can be used to show that a language is regular. It follows that a language is regular if **the set of distinguishable classes of strings in the language is finite i.e. (n finite number of classes) and the FA accepting L must have at least n number of states.**

So, given a language L, Myhill_Neorde theorem can be used directly to build a very simple FA that can accept all strings in L.

Example 3:
For us to build an FA for language odd-odd defined over £ = {0,1}, we can find the number of states of the FA using Myhill-Nerode theorem.

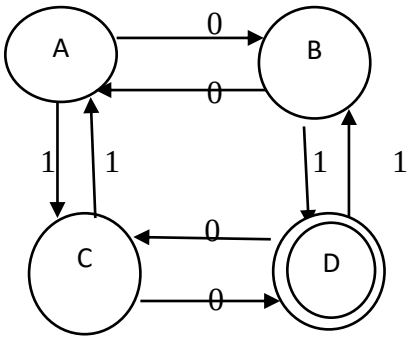
The language odd-odd, divide all its strings into the following distinguishable classes:

1. String containing even number of 0 and even number of 1

2. String containing odd number of 0, even number of 1
3. String containing even number of 0, odd number of 1
4. String containing odd number of 0, odd number of 1

Therefore, the FA accepting the language will contain 4 states each representing one of the classes identified.

Drawing the FA by representing each class with a corresponding state.



State A: even 0 and even 1
State B: odd 0 and even 1
State C: odd 1 and even 0
State D: odd 0 and odd 1

Since we are able to draw an FA for the language consisting of number of states equal to the number of distinguishable classes in the language, then the language odd-odd is regular.

Note: Myhill-Nerode Theorem can be sufficiently used to prove that a language is regular.

4.0 Summary

In this unit, we had discussed that :

- Two strings x and y are said to be distinguishable with respect to a language L, if there exists a string *z* such that: $xz \in L$ and yz *is not* $\in L$ or vice versa
- Two strings x and y are said to be indistinguishable with respect to a language L, if there exists a string *z* such that: $xz \in L$ and yz *is* $\in L$ or vice versa
- Myhill-Nerode Theorem states that a language is regular if the set of distinguishable classes of strings in the language is finite i.e. (n finite number of classes) and the FA accepting L must have at least n number of states.

- To draw an FA directly from a given language, the strings in **£*** is partitioned into distinguishable classes with respect to the language and the FA is built by representing each class with a corresponding state.
- Myhill-Nerode Theorem is sufficient to prove the regularity of a language.

5.0 Self-assessment Questions

1. What do you understand by distinguishable string with respect to a Language
2. State Myhill-Nerode Theorem
3. Build an FA for the language L = **{** bw: w ϵ □*, □□□□□a□b□**}** using the Myhill-Nerode Theorem

6.0 Tutor-marked assignment

Given a language **L = { bw | cw: w ϵ □ *}** defined over an alphabet **□ = {a, b, c}**, build an FA accepting L using Myhill-Nerode Theorem

7.0 References and Further Reading

Mishra, K.L.P. & Chandrasekaran, N. (2008). Theory of Computer Science: Automata, Languages and Computation (3rd Edition). Prentice-Hall of India (PHI), New Delhi, India

Martin, J.C.(2011). Introduction to Languages and The Theory of Computation (4th Edition). McGraw-Hill, New York, USA.

Sipser, M. (2012) Introduction to the Theory of Computation (3rd Edition). Cengage Learning, Boston, USA.

Tourlakis, G. (2012). Theory of Computation (1st Edition). Wiley, USA

https://www.cs.odu.edu/~toida/nerzic/390teched/regular/reg-lang/Myhill-Nerode/index.html

Module 4 FSM with Output

Unit 1 Introduction to More Machines

Unit 2 Introduction to Mealy Machines

Unit 1: More Machines

1.0 Introduction

This unit introduces you to the Moore machine an FSM with output, focusing on its formal description. Its transition table and diagram are also described with examples.

2.0 Learning Outcomes

Upon completion of this unit, you should be able to:

i. Give formal definition of a Moore machine
ii. Build a Moore machine from a given transition table
iii. Compute the output of given input to a Moore machine

3.0 Main Content

3.1 Moore Machines(10Mins) [SAQ1]

Recalls that we have considered so far finite state machine with binary output i.e. either they accept the string or they do not accept the string. This acceptability was decided on the basis of the reachability of final state from the initial state. Now, we remove this restriction and consider other models such as Moore machine model where the outputs can be chosen from some other alphabets.

Definition: Formally a Moore machine is a 6-tuple $(Q, \Sigma, \Gamma, \delta, \Lambda, q0)$

Q is a finite set of state

Σ *is the input alphabet*

Γ *is the output alphabet*

δ *is transition function* $\Sigma \times Q \longrightarrow Q$

Λ *is the output function* $Q \longrightarrow \Gamma$

q0 is a distinct member of Q representing the initial state

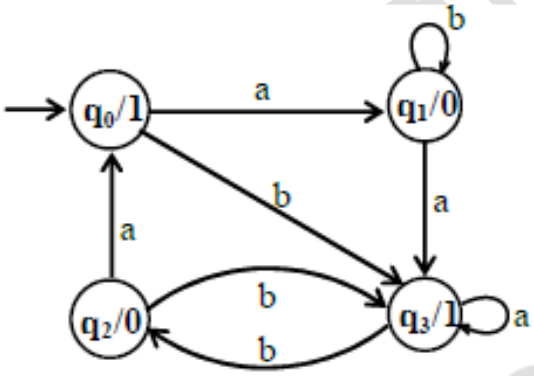
Note: in Moore machine the output function depends only on the present state and is independent of the current input. Note also that no state is designated to be final state.

[SAQ2,3]

Example 1: Consider the following Moore machine define over $\Sigma = \{a,b\}$ and $\Gamma = \{0,1\}$ represented with transition table.

State	Next state (δ)		output
	a	b	
q_0	q_1	q_3	1
q_1	q_3	q_1	0
q_2	q_0	q_3	0
q_3	q_3	q_2	1

The transition table is given below:



Note that the states are labelled along with the output alphabet to be printed when each state is entered. so there is no question of accepting any language by Moore machine. Moreover, the state to be initial is not important as if the machine is used several times and is restarted after some time, the machine will be started from the state where it was left off.

Now, running the string aaabbbaabba over the machine, the corresponding output string will be 101101010001. It is to be noted that the length of the output string is more than the input string as the initial state print extra character 1

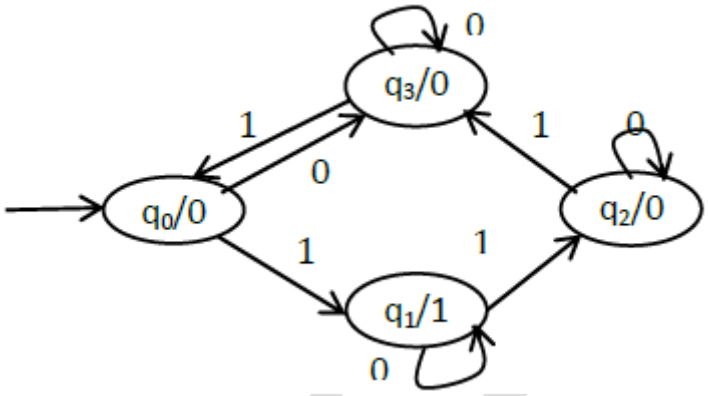
Example 2: Given below by the transition table is a Moore machine over £ = {0,1} and Γ ={0,1}

State δ

	0	1	Λ

q_0	q_3	q_1	0
q_1	q_1	q_2	1
q_2	q2	q_3	0
q_3	q_3	q_0	0

The transition diagram drawn from the table is shown below:



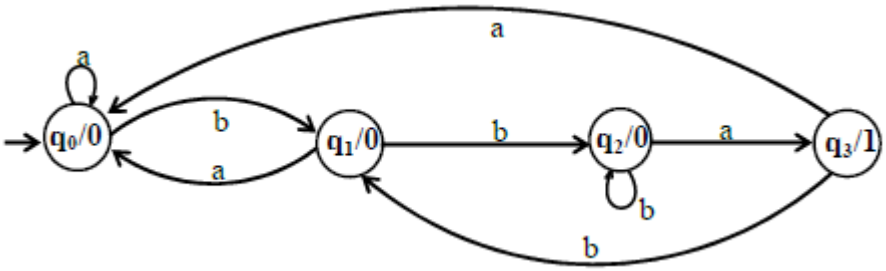
What will be the output string if the other input string 0111 is read?

$0q_0 \xrightarrow{0} 0q_3 \xrightarrow{1} 0q_0 \xrightarrow{1} 1q_1 \xrightarrow{1} 0q_2$

Output = 00010

In some cases, the relationship between an input string and the corresponding output string may be identified by the Moore machine.

Example 3: consider the following Moore machine:



If the string bbababababbaba is read, the output string will be:

Input		b	b	b	a	b	a	a	b	b	a	a
State	q_0	q_1	q_2	q_2	q_3	q_1	q_0	q_0	q_1	q_2	q_3	q_0
output	0	0	0	0	1	0	0	0	0	0	1	0

output: 0001000000100.

From the machine it shows that immediately state q3 is reached, the machine will print 1 and to enter q3 starting from q0, the string must contain bba. Thus, the number of 1's in the output string will be the same as the number of sub-string bba that occurs in the corresponding input string (2 in this case).

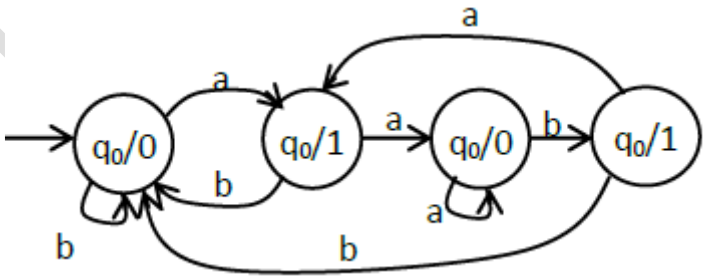
Example 4: Question-Design a Moore machine which counts the occurrence of substring aab in input string.

Solution: Let the Moore machine be
M0 = (Q, Σ, Δ, δ, λ,q0)
Q = { q0, q1, q2, q3 }
Σ = { a, b}
We will design this machine in such a way that machine prints out the character 0 except for q3, which prints a 1. To get q3, we must come from state q2 and have 1st read b. To get state q2, we must read at least two a's in a row, having started in any state.

Δ = { 0 , 1 }

λ'(q0) = 0, λ'(q1) = 0, λ'(q2) = 0, λ'(q3) = 1

The following machine will count aab occurrence for us.



So, it can be generalized that Moore machine has the additional power to count other than recognize.

4.0 Summary

In this unit, you have learnt that:

- A Moore machine is a 6-tuple (Q, £, Γ, δ, Λ, q0).
- The output of a Moore Machine depends only on the current state of the machine
- Moore machine has the power to recognize and count.

5.0 Self-Assessment Questions

i. Define a Moore machine

ii. Build a Moore machine from a given transition table:

Present State	Next State		Output
	a = 0	a = 1	(λ)
⟶q1	q1	q2	0
q2	q1	q3	0
q3	q1	q3	1

iii. Compute the output the Moore machine in ii after reading the string 100100

6.0 Tutor-Marked Assignment

1. A Moore machine is described by the following transition table:

Present State	Next State		Output
	a = 0	a = 1	(λ)
⟶q1	q3	q20	1
q20	q1	q40	0
q21	q1	q41	1
q3	q21	q1	0
q40	q41	q3	0
q41	q41	q3	1

What is the output of the machine after reading the string 11010001110?

7.0 References and Further Reading

Mishra, K.L.P. & Chandrasekaran, N. (2008). Theory of Computer Science: Automata, Languages and Computation (3rd Edition). Prentice-Hall of India (PHI), New Delhi, India

Anderson, J.A. (2012). Modern Applications of Automata Theory (1st Edition). Cambridge University Press, London.

Tourlakis, G. (2012). Theory of Computation (1st Edition). Wiley, USA

https://www.javatpoint.com/automata-moore-machine

https://www.geeksforgeeks.org/mealy-and-moore-machines-in-toc/

Unit 2: Introduction to Mealy Machines

1.0 Introduction

This unit introduces you to the Mealy machine as an FSM with output, focusing on its formal description and its peculiar characteristics that differentiate it from Moore machines. Examples on how to build Mealy machine are also given and explained to you.

2.0 Learning outcomes

At the end of this unit, you should be able to:

i. Formally define a Mealy machine
ii. Construct a Mealy machine from a given transition table
iii. Compute the output of Mealy machine from a given input string

3.0 Main Content

3.1 Mealy Machine(5Mins) [SAQ1]

Definition: Formally a Mealy machine is a 6-tuple (Q, £, Γ, δ, Λ, q0)
Q is a finite set of state
£ is the input alphabet
Γ is the output alphabet
δ is transition function £ × Q → Q
Λ is the output function Q → Γ
q0 is a distinct member of Q representing the initial state

Note: in Mealy machine the output function depends on both the present state and the current input (i.e. each transition in the machine has a specific output). Note also that no state is designated to be final state.

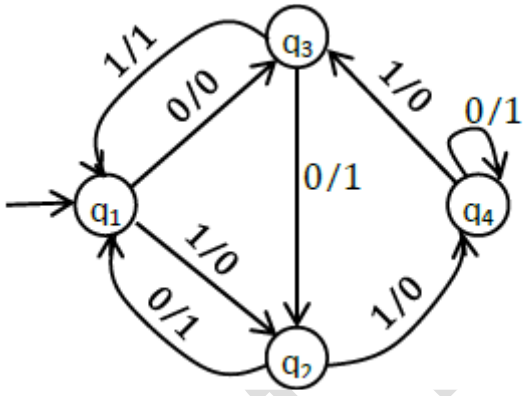
[SAQ1]

Example 1: Given the transition table below representing a Mealy machine defined over £ = {0,1} and Γ = {0,1}

	Input state	Output	state	output

q_0	q_3	0	q_2	0
q_1	q_1	1	q_4	0
q_2	q2	1	q_1	1
q_3	q_4	1	q_3	0

The transition diagram is shown below:

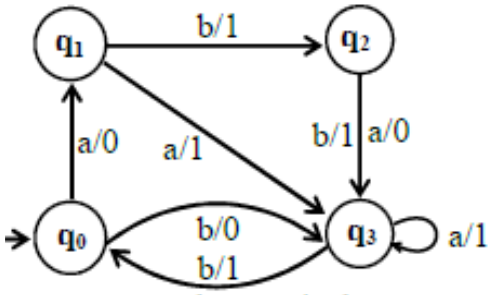


Given an input string 0011 to be read:

→q1 $\xrightarrow{0/0}$ q3 $\xrightarrow{0/1}$ q2 $\xrightarrow{1/0}$ q4 $\xrightarrow{1/0}$ q3

The output will be 0100. It may be noted that in Mealy machine, the length of output string is equal to that of input string.

Example 2: Consider the Mealy machine shown below, having the states q0, q1, q2, q3 , where q0 is the start state and £ = {a,b}, Γ={0,1}

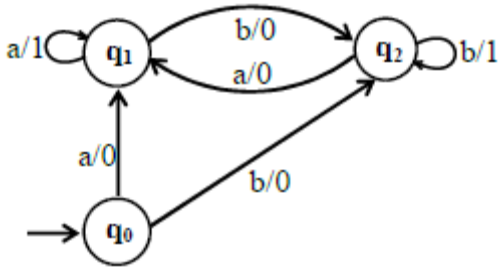


Running the string abbabbba over the above machine, the corresponding output string will be 11011010, which can be determined by the following table as well:

Input		a	b	b	a	b	b	b	a
States	q_0	q_1	q_2	q_3	q_3	q_0	q_3	q_0	q_1
output		0	1	1	1	1	0	1	0

Apart from producing outputs, Mealy machine can be used to carry out some computations on the input strings.

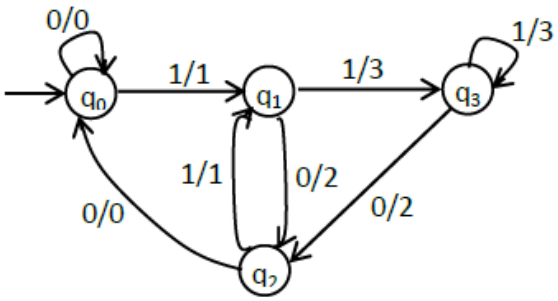
Example 3: Consider the following Mealy machine having the states q0, q1, q2 , where q0 is the start state and $\mathcal{L} = \{a,b\}$, $\Gamma=\{0,1\}$



It can be observed that in the above Mealy machine, if in the output string the n^{th} character is 1, it shows that the n^{th} letter in the input string is the second in the pair of double letter, thereby computing the position of a repeating characters in the input string.
For babaababba as input string the machine will print 0000100010.

Example 4: Question- Construct a mealy machine which calculate residue mod-4 for each binary string treated as binary integer.

Solution:
M0 = (Q, Σ, Δ, δ, λ,q0)
Q = { q0, q1, q2, q3 }
Σ = { 0, 1}
Δ = { 0, 1, 2, 3}
The machine is given below:



4.0 Summary

In this unit, you hae learnt that:

- A Mealy machine is a 6-tuple (Q, £, Γ, δ, Λ, q0).
- The output of a Mealy Machine depends on the current state of the machine and the current input symbol.
- Mealy machine has the power to recognize and count ad perform some simple computations.

.

5.0 Self-Assessment Questions

1. What is a Mealy machine?
2. Given the transition function below, Construct a corresponding Mealy machine:

$\lambda'(q_1, 0) = 0$	$\lambda'(q_1, 1) = 0$
$\lambda'(q_2, 0) = 1$	$\lambda'(q_2, 1) = 0$
$\lambda'(q_3, 0) = 1$	$\lambda'(q_3, 1) = 1$
$\lambda'(q_4, 0) = 1$	$\lambda'(q_4, 1) = 0$

3. Compute the output of the Mealy machine built in 2 above

5.0 Tutor-Marked Assignment

Design a mealy machine, which prints 1's complement of input bit string over alphabet $\Sigma = \{0, 1\}$.

7.0 References and Further Reading

John, E.H, Rajheev, M. & Jeffrey, D.U. (2001). Introduction to Automata Theory, Languages and Computation (2nd Edition). Addison-Wesley, Pearson Education, USA.

Martin, J.C.(2011). Introduction to Languages and The Theory of Computation (4th Edition). McGraw-Hill, New York, USA.

Anderson, J.A. (2012). Modern Applications of Automata Theory (1st Edition). Cambridge University Press, London.

https://www.geeksforgeeks.org/mealy-and-moore-machines-in-toc/

https://www.geeksforgeeks.org/difference-between-mealy-machine-and-moore-machine/

Module 5 Grammar and Push Down Automata

Unit 1 Introduction to Grammar

Unit 3 Push Down Automata

Unit 1: Overview of loaders and linkers

1.0 Introduction

This unit introduces you to the concepts Grammar as used in formal Languages. Its formal definition, strings derivation and classification. You are also presented with some examples to boost your understanding of the concepts.

2.0 Learning Outcomes

After completing this unit, you should be able to:

i. Define a Grammar
ii. Derive strings from a grammar
iii. Classify grammar into respective hierarchies

3.0 Grammar (10Mins) [SAQ1]

According to English Dictionary "grammar is the study of science of rules for the combination of words into sentences in a language". For example, in common English, the following sentences is valid

a. Ade is a boy.
b. Ade goes to school everyday

Whereas, the two examples below are not valid

a. A ade is boy
b. Ade is going to school everyday

Formally Grammar is defined to be a quadruple

G = (S, N, T, P)

Where:

S is the *start symbol*
N is a set of *non-terminal symbols*
T is a set of *terminal symbols*
P is a set of *productions* or *rewrite rules* ($P : N \rightarrow N \cup T$)

Given a grammar below

1. <goal> ::= <expr>
2. <expr> ::= <expr> <op> <term>
3. | <term>
4. <term> ::= number
5. | id
6. <op> ::= +
7. | -

This grammar defines simple expressions with addition and subtraction over the inputs *id* and *number*

A distinguished non-terminal symbol from which all strings of a language are derived is called **Start Symbol (S)**

Those symbols appearing as a left part of the rules enclosed in in corner bracket are called **Non-Terminals (N).** Those in symbols of the right part of the rule which are not enclosed in corner bracket are the **Terminal Symbols (T)**

A Production rule (P) is an ordered pair (U, X) written as U →X, or U ::=X where U is a non-terminal symbol and X is a non-empty finite string of non-terminal and terminal symbols

Example:
S = < *goal*>
T = number, id, +, -
N = <*goal*>, <*expr*>, <*term*>, <*op*>
P = 1, 2, 3, 4, 5, 6, 7

This format which uses corner bracket (<> for non-terminals) and **::=** (for derivation) is called ***Backus-Naur form*** **(BNF):** a kind of notation or meta language to describe grammar that is used to specify the syntax of programming language formally.

In another format capital Roman letters are used to represent non-terminals while small letters are used for terminals. The symbol → is used to represent production.

The above Grammar can be written in this format as:

G → E

E → EOT | T

T → number | id

O → + | -

Given a grammar, valid sentences can be derived by repeated substitution (re-writing the non-terminals into another form by applying one or more of the rules) until all non-terminals are

replaced with terminals. This process is called **Derivation.** For example using the above grammar, the String x + 2 – y can be derived as shown below:

Prod'n.	Result
	<goal>
1	<expr>
2	<expr> <op> <term>
5	<expr> <op> y
7	<expr> - y
2	<expr> <op> <term> - y
4	<expr> <op> 2 - y
6	<expr> + 2 - y
3	<term> + 2 - y
5	x + 2 - y

x + 2 – y is a sentence derived from the grammar above.

3.1 Types of Derivation [SAQ2]

1. **Rightmost derivation:** in this type of derivation, the rightmost non-terminal is substituted in each derivation step.

Example: Given the grammar G below

1. S -------> AB
2. A -------> A
3. | a
4. B -------> Bb
5. |b

Show that the string abbbb can be derived from G.

Production	Result
	S
1	AB
4	ABb
4	ABbb
4	ABbbb
5	Abbbb
3	abbbb

2. **Leftmost Derivation:** here the leftmost non-terminal is replaced in each derivation step. Using the same example above

Production	Result
	S
1	AB
3	aB
4	aBb
4	aBbb
4	aBbbb
5	abbbb

3.2 Sentential Form and Sentences

A **sentence** consists of set of terminals derivable from a grammar. For example abbbb is a sentence derived from our grammar G above.

A **Sentential form** is sentence that contains non-termianals and terminals derivable from a grammar. E.g. aBbb is a sentential form derived from Grammar G.

In general, we write:

$\mathbf{W_1}$ -------> * $\mathbf{W_n}$

if $\mathbf{W_1}$ -------> $\mathbf{W_2}$ -------> $\mathbf{W_3}$…….. -------> $\mathbf{W_n}$

It means that the string Wn can be derived from the String W1 by **zero or more** application of the production rule of the grammar

It is always the case that: **W** -------> * **W**

e.g. AB -------> * aBbbb , AB -------> * AB from grammar G above

Also,

W1 -------> $^{+}\mathbf{W_n}$ means the string W_n can be derived from the string W_1 by **one or more** application of production of the grammar.

e.g. AB -------> $^{+}$ aBbb

3.3 Classification of Grammar [SAQ3]

Chomsky classified grammars into 4 hierarchies which consists of the following levels:

- **Type-0 grammars:** These grammars have rules of the form **α -------> β** , Where **α and β ϵ (N ∪ T)** strings of terminals and/or non-terminals with no restriction.. They include all formal grammars. These languages are also known as the recursively enumerable languages. They generate exactly all languages that can be recognized by a Turing machine.

Example:

$S \rightarrow ABS$

$S \rightarrow \varepsilon$
$BA \rightarrow AB$
$BS \rightarrow b$
$Bb \rightarrow bb$
$Ab \rightarrow ab$
$Aa \rightarrow aa$

- **Type-1 grammars:** These grammars have rules of the form

  α A β -------> α ϒ β

  With **A ϵ N** non-terminal and **α, βand ϒ ϵ (N ∪ T)** strings of terminals and/or non-terminals. The strings **α** and **β** may be empty, but **ϒ** must be nonempty. The rule **S ----> ε** is allowed if **S** does not appear on the right side of any rule.

  This type of grammar is called **context-sensitive grammar**. The languages described by these grammars are exactly all languages that can be recognized by a **linear bounded automaton**

  Example:

  $S \rightarrow AB$
  $A \rightarrow aB$
  $B \rightarrow ABb$
  $Bb \rightarrow bb$
  $Ab \rightarrow ab$

- **Type-2 grammars** : These are defined by rules of the form:

  A-------> ϒ ,

  With **A ϵ N** a nonterminal and **ϒ ϵ (N ∪ T)** a string of terminals and/or non-terminals.

  They are called context-free grammars. These grammars generate languages that can be recognized by a **non-deterministic pushdown automaton.** Context-free languages are the theoretical basis for the phrase structure of most programming languages.

  Example:

$S \rightarrow AB$
$A \rightarrow abB$
$B \rightarrow aaB$
$B \rightarrow b$

- **Type-3 grammars**: they are defined by rules of the forms:

  A-------> a, and

  A -------> a B

  This type of grammar restricts its rules to a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single non-terminal **(right regular).** Alternatively, the right-hand side of the grammar can consist of a single terminal, possibly preceded by a single nonterminal **(left regular)** The rule **S ----> ε** is also allowed here if **S** does not appear on the right side of any rule. They are called **Regular Grammar**. These grammars generate languages that can be recognized by a **finite state automaton**. Additionally, this family of formal languages can be obtained by **regular expressions.** Regular languages are commonly used to define the lexical structure of programming languages.

  Example:

  $S \rightarrow aA$
  $S \rightarrow \varepsilon$
  $A \rightarrow bB$
  $B \rightarrow a$

4.0 Summary

In this unit, you have learnt that:

- Formally Grammar is defined to be a quadruple **G = (S, N, T, P)**
- Repeated substitution of rules (re-writing the non-terminals into another form by applying one or more of the rules) until all non-terminals are replaced with terminals is called **Derivation**
- There are two ways to derive a string from a grammar: Rightmost and Leftmost Derivation
- A Sentential form is sentence that contains non-termianals and terminals derivable from a grammar. E.g. aBbb is a sentential form derived from Grammar G.

- Chomsky classified grammar into four hierarchies: type 0 (recursively enumerable), type 1 (Context sensitive), type 2(Context free) and type 3 (Regular)

5.0 Self-assessment Questions

i. What is a Grammar

ii. Derive two strings from the grammar given below

S -> F
S -> (S + F)
F -> a

iii. Explain the four classes of grammar with examples

6.0 Tutor-marked assignment

consider the grammar G2 defined as follows:

<print Statement> -------> PRINT | PRINT <expr>
<expr> -------> <var> | <var>,<exp>
<var> -------> <letter> | <letter> <digit>
<letter> -------> A|B|……|Y|Z
<digit> -------> 0|1|…….|8|9.

Show that the following Strings can be derived from G2.

1. PRINT 2. PRINT M4 3. PRINT Y8, Y

7.0 References and Further Reading

Mishra, K.L.P. & Chandrasekaran, N. (2008). Theory of Computer Science: Automata, Languages and Computation (3rd Edition). Prentice-Hall of India (PHI), New Delhi, India

Martin, J.C.(2011). Introduction to Languages and The Theory of Computation (4th Edition). McGraw-Hill, New York, USA.

Sipser, M. (2012) Introduction to the Theory of Computation (3rd Edition). Cengage Learning, Boston, USA.

Unit 2: Push Down Automata

1.0 Introduction

In this unit, your attention is drawn to the concept of Push Down Atomaton (PDA) a more powerful automaton compared to FA's. forma describtion of the machine, its special moves, string acceptance and types are discussed.

2.0 Learning Outcomes

After completing this study unit, you should be able to:

i. Formally define a Push Down Automata (PDA)
ii. Trace the move of a PDA while reading an input string
iii. Describe a Non-Deterministic PDA

3.0 Push Down Automata(7Mins) [SAQ1]

Recall that the major drawback of finite automaton is its limited memory and absence of mechanism to remember the count of input symbol and to match.
Let us consider the language **L = {= $0^n10^n \geq 1$}**

Be aware that a finite automaton cannot accept L, because it has to remember the number of 0's in a string and so it will require an infinite number of states. This difficulty can be avoided by adding an auxiliary memory in the form of a stack. The kind of machine with finite number of state and infinite auxiliary memory in form of a stack is called PDA. A machine with mechanism to remember, count and match.

To implement the language L in a PDA, the 0's in the given string are added to the stack. Consecutively, when a symbol 1is encountered, 0 is removed from the stack, thus the matching of numbers of 0's and 1's is done. An empty stack after consumption of the string indicates acceptance of the string by the PDA.

Definition:
Formally a PDA is a 7-tuple (Q,£,r,δ,Λ,q0,z0,F) where
Q is a finite set and an empty set of states
£ is a finite, non-empty set of input alphabet
r is a set of stack symbols
δ is transition function which maps from Q × (£ U {Λ}) × r to Q × r (finite subset)*
q0 ϵ Q is the initial state
q0 ϵ r is the initial stack symbol
F C Q is the set of final states

3.0 Moves of pushdown automata [SAQ2]

The moves of PDA means that what are the options to proceed further after reading input in some state and counting some symbols on the stack. The move is of two types:

i. In the first type, an input symbol is read and depends upon the topmost symbol on the stack and the present state, the PDA has number of choices to proceed.
Example: $\delta (q, a, z)1— \{(q, \beta)\}, q \epsilon Q, \beta \epsilon r^*$

ii. In the second type of move, the input symbol is not read, and the topmost of the stack is modified. This is also known as Λ-moves.

Example: the PDA accepting language $a^n b^n$

Move number	State	output	Stack symbol	moves
1	q_3	a	z_2	$(q_1, a\ q_1)$
2	q_1	a	a	(q_1, aa)
3	q_1	b	a	(q_2, Λ)
4	q_2	b	a	(q_1, Λ)
5	q_2	Λ	z_0	(q_1, z_0)

Tracing the moves of the machine for the string aabb, will yield:

Move #	Result stack	Unread input	Stack
—	q_0	aabb	z_0
1	q_1	abb	az_0
2	q_1	bb	aaz_0
3	q_2	b	az_0
4	q_2	Λ	z_0

5	q_3	—	z_0

3.2 Acceptance of String By PDA

A string is accepted by PDA if

1. δ (q0 ,w,z0)
2. Λ (q0 ,w,z0)
   *(F, Λ, δ), where Y ϵ r *(Q, Λ, z0).

Example: a PDA accepting the language L = { $a^n b^m$ | n ≥ m}

Move number	State	Input	Stack symbol	moves
1	q_0	a	z_0	(q_1, a z_1)
2	q_0	b	z_0	(q_1, b z_0)
3	q_1	a	a	(q_1, aa)
4	q_1	a	b	(q_1,Λ)
5	q_1	b	a	(q_1,Λ)
6	q_1	b	b	(q_1, bb)
7	q_1	Λ	a	(q_+, a)

Trace the moves of the machine for the string aaabb

Move #	Result stack	Unread input	Stack
—	q_0	aaabb	z_0
1	q_1	aabb	az_0
3	q_1	abb	aaz_0
3	q_1	bb	aaaz_0
5	q_1	b	az_0

5	q_1	Λ	az_0
7	q_f	—	z_0

Accepted by reaching final state for aaaabb, aaab, aabb.

3.3 Non-Deterministic PDA[SAQ3]

A Non deterministic PDA has finite number of choices for its inputs i.e there are more than oone moves for at least one input configuration of the PDA.

A NPDA accepts an input if a sequence of choices leads or presence of n-moves to some final state or cause PDA to empty its stack.

Example: the following PDA accepting language Palindrome

Move #	Result stack	Unread input	Stack symbol	Moves(s)
1	q_0	a	z_0	$(q_0, a z_0)$, $(q_0, a z_0)$
2	q_0	a	a	(q_0, aa), (q_1, a)
3	q_0	a	b	(q_0, ab), (q_1, b)
4	q_0	b	z_0	(q_0, bz_0), (q_1, b)
5	q_0	b	a	(q_0, ba), (q_1, a)
6	q_0	b	b	(q_0, bb), (q_1, b)
7	q_0	Λ	z_0	(q_0, z_0)
8	q_0	Λ	a	(q_0, a)
9	q_0	Λ	b	(q_0, b)
10	q_1	a	a	(q_0, Λ)
11	q_1	b	b	(q_0, Λ)
12	q_1	Λ	z_0	(q_0, z_0)

4.0 Summary

In this unit you have learnt that:

- PDA is a kind of machine with finite number of state and infinite auxiliary memory in form of a stack
- A PDA is formally defined as a 7-tuple (Q,£,r,δ,Λ,q0,z0,F)
- A PDA can make two kinds of moves:

    i. an input symbol is read and depends upon the topmost symbol on the stack and the present state, the PDA has number of choices to proceed.

    ii. the input symbol is not read, and the topmost of the stack is modified..

- In a Non deterministic PDA, there are more than one moves for at least one input configuration of the PDA.

5.0 Self-assessment Questions

i. Define a PDA
ii. Describe the Moves of a PDA
iii. Describe a Non-Deterministic PDA

6.0 Tutor-marked assignment

Given a PDA represented by the transition function:
$\delta (q0, a, Z0) \vdash (q1, aZ0)$
$\delta (q1, a, a) \vdash (q1, aa)$
$\delta (q1, b, a) \vdash (q2, a)$
$\delta (q2, b, a) \vdash (q2, a)$
$\delta (q2, a, a) \vdash (q3, \wedge)$
$\delta (q3, a, a) \vdash (q3, \wedge)$
$\delta (q3, \wedge, Z0) \vdash (qf, Z0)$
Trace the moves for the input strings: aaabbbbaaa, abbbba, abbaa.

7.0 References and Further Reading

John, E.H, Rajheev, M. & Jeffrey, D.U. (2001). Introduction to Automata Theory, Languages and Computation (2nd Edition). Addison-Wesley, Pearson Education, USA.

Martin, J.C.(2011). Introduction to Languages and The Theory of Computation (4th Edition). McGraw-Hill, New York, USA.

Anderson, J.A. (2012). Modern Applications of Automata Theory (1st Edition). Cambridge University Press, London.

Sipser, M. (2012) Introduction to the Theory of Computation (3rd Edition). Cengage Learning, Boston, USA.