Debre Tabor University

Technology Faculty

Department of Computer Science

Module:

Automata and Complexity Theory

Compiled by Getasew N. and

Moges T.

Debre Tabor,

Ethiopia

2014 E.C

Introduction

Computer Science is a practical discipline. Those who work in it often have a marked preference for useful and tangible problems over theoretical speculations. This is certainly true of computer science students who are interested mainly in working on difficult applications from the real world. Theoretical questions are interesting to them only if they help in finding good solutions. This attitude is appropriate since without applications there would be little interest in computers. But given this practical orientation one might well ask "**why study theory**?"

The first answer is that theory provides concepts and principles that help us understand the general nature of the discipline. The field of computer science includes a wide range of special topics, from machine design to programming. The use of computers in the real world involves a wealth of specific details that must be learned for a successful application. This makes computer science a very diverse and broad discipline. But in spite of this diversity, there are some common underlying principles. To study these basic principles we construct abstract models of computers and computation. These models embody the important features that are common to both hardware and software and that are essential to many of the special and complex constructs we encounter while working with computers. Even when such models are too simple to be applied immediately to real-world situations, the insights we gain from studying them provide the foundations on which specific development is based. This approach is of course not unique to computer science. The construction of model is one of the essentials of any scientific discipline, and the usefulness a discipline is often dependent on the existence of simple, yet powerful, theories and laws.

A second, and perhaps, not so obvious answer, is that the ideas we will discuss have some immediate and important applications. The field of digital design, programming languages, and compilers are the most obvious examples, but there are many others. The concept we study here runs like a thread through much of computer science, from operating systems to pattern recognition.

The third answer is one of which we hope to convince the reader. The subject matter is intellectually stimulating and fun. It provides many challenging, puzzle-like problems that can lead to some sleepless nights. This is problem-solving in its pure essence.

[**An Introduction to Formal languages and Automata, Peter Linz, Third Edition**]

CHAPTER ONE:

1. Introduction to Formal Language and Automata Theory

Unit Contents:

1.1.Introduction to Formal Languages

Formal Language: is an abstraction of the general characteristics of programming languages. It consists of a set of symbols and rules of formation by which symbols combined to sentences. It is the set of all strings permitted by the rules of formation and has some essential features of programming language.

N**atural Language vs. Formal Language**

Natural Language: Written and/or spoken languages in the world, such as Chinese, English, Japanese, Amharic, etc.

- Syntax
- Semantics

Formal Language: A language specified by a well -defined set of rules of syntax. A study of formal languages is important to computer science. For example, we need to understand what kind of **statements** are acceptable in the C programming language. This is the task of a compiler of a programming language.

Why study formal language?

Useful: Amide at constructing abstract models of features that is common to both hardware and software. Applicable in the fields of digital design, programming and compilers design. The concepts have significant importance from operating systems to pattern recognition. It provides concepts and principles that help us understand the general nature of science.

Connected: Too many other branches of knowledge.

Rigorous: The subject matter is intellectually stimulating and fun with many puzzles.

Stable: The basics have not changed much in the last three decades.

1.2.Introduction to Automata Theory

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

Automata enable scientists to understand how machines compute the functions and solve problems. The main motivation behind developing the Automata Theory was to develop methods to describe and analyze the dynamic behavior of discrete systems. Automata is originated from the word "Automaton" which is closely related to "Automation".

1.3.Mathematical Preliminaries and Notation

- Set Theory
- Relations
- Functions

a) Set Theory:

A **Set** is an unordered collection of objects, known as elements or members of the set. An element 'a' belong to a set A can be written as 'a ∈ A', 'a ∉ A' denotes that a is not an element of the set A.

Representation of a Set

A set can be represented by various methods. 3 common methods used for representing set:

1. Statement form.
2. Roaster form or tabular form method.
3. Set Builder method.

Statement form

In this representation, the well-defined description of the elements of the set is given. Below are some examples of the same.

1. The set of all even numbers less than 10.
2. The set of the numbers less than 10 and more than 1.

Roster form

In this representation, elements are listed within the pair of brackets {} and are separated by commas. Below are two examples.

1. Let N is the set of natural numbers less than 5.

N = { 1 , 2 , 3, 4 }.

2. The set of all vowels in the English alphabet.

V = { a , e , i , o , u }.

Set builder form

In Set-builder set is described by a property that its member must satisfy.

1. {x: x is even number divisible by 6 and less than 100}.
2. {x: x is natural number less than 10}.

Equal sets

Two sets are said to be equal if both have same elements. For example A = {1, 3, 9, 7} and B = {3, 1, 7, 9} are equal sets.

NOTE: Order of elements of a set doesn't matter.

Subset

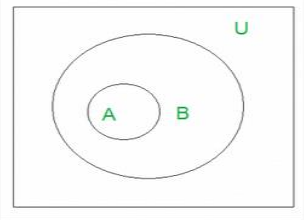
A set A is said to be a **subset** of another set B if and only if every element of set A is also a part of other set B.

Denoted by '⊆'.

'A ⊆ B ' denotes A is a subset of B.

To prove A is the subset of B, we need to simply show that if x belongs to A then x also belongs to B.

To prove A is not a subset of B, we need to find out one element which is part of set A but not belong to set B.



'U' denotes the universal set.

Above Venn Diagram shows that A is a subset of B.

Size of a Set

The size of a set can be finite or infinite.

For example

Finite set: Set of natural numbers less than 100.

Infinite set: Set of real numbers.

Size of the set S is known as **Cardinality number**, denoted as |S|.

Example: Let A be a set of odd positive integers less than 10.

Solution : A = {1,3,5,7,9}, Cardinality of the set is 5, i.e., |A| = 5.

Note: The cardinality of a null set is 0.

Proper Subset: If S1 C S, but S contains an element, not in S1, we can say that S1 is a proper subset of S; we write this as S1 C S.

Power Sets

The power set is the set all possible subset of the set S. Denoted by P(S).

Example: What is the power set of {0,1,2}?

Solution: All possible subsets

{∅}, {0}, {1}, {2}, {0,1}, {0,2}, {1,2}, {0,1,2}.

Note: Empty set and set itself is also a member of this set of subsets.

The cardinality of the power set is 2^n, where n is the number of elements in a set.

Cartesian Products

Let A and B be two sets. Cartesian product of A and B is denoted by A × B, is the set of all ordered pairs (a,b), where a belong to A and b belong to B.

A × B = {(a, b) | a ∈ A ∧ b ∈ B}.

Example 1. What is Cartesian product of A = {1,2} and B = {p, q, r}. Solution : A × B ={(1, p), (1, q), (1, r), (2, p), (2, q), (2, r) };

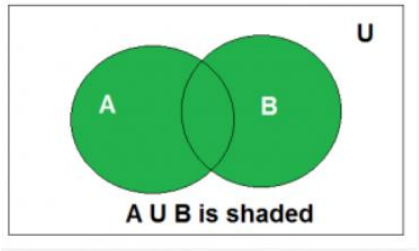
The cardinality of A × B is N*M, where N is the Cardinality of A and M is the cardinality of B. The notation is extended in an obvious fashion to the Cartesian product of more two sets; generally S1 X S2 X ... X Sn= {(X1, X2, X3… Xn):Xi ϵ Si}

Note: A × B is not the same as B × A.

Set Operations

Union

Union of the sets A and B, denoted by A ∪ B, is the set of distinct element belongs to set A or set B, or both.



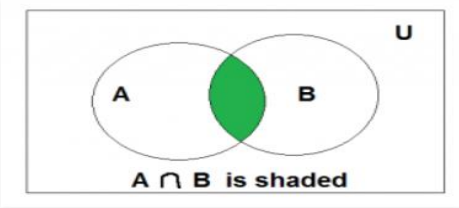
Above is the Venn Diagram of A U B.

Example : Find the union of A = {2, 3, 4} and B = {3, 4, 5};

Solution : A ∪ B = {2, 3, 4, 5}.

Intersection

The intersection of the sets A and B, denoted by A ∩ B, is the set of elements belongs to both A and B i.e. set of the common element in A and B.



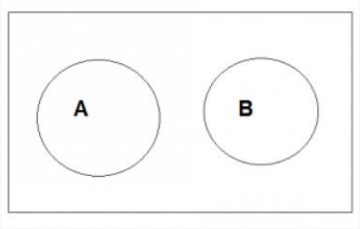
Above is the Venn Diagram of A ∩ B.

Example: Consider the previous sets A and B. Find out A ∩ B.

Solution : A ∩ B = {3, 4}.

Disjoint

Two sets are said to be disjoint if their intersection is the empty set .i.e. sets have no common elements.



Above is the Venn Diagram of A disjoint B.

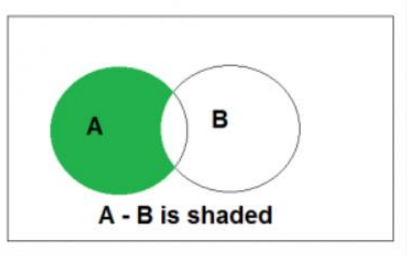
For Example

Let A = {1, 3, 5, 7, 9} and B = { 2, 4 ,6 , 8} .

A and B are disjoint set both of them have no common elements.

Set Difference

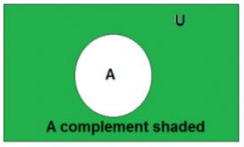
The difference between sets is denoted by 'A – B', is the set containing elements of set A but not in B. i.e all elements of A except the element of B.



Above is the Venn Diagram of A-B.

Complement

A complement of a set A, denoted by A^C, is the set of all the elements except A. Complement of the set A is U – A.



Above is the Venn Diagram of A^c

Formula:

- $A \cup B = n(A) + n(B) - n(A \cap B)$
- $A - B = A \cap B'$

Properties of Union and Intersection of sets:

- **Associative Properties:** $A \cup (B \cup C) = (A \cup B) \cup C$ and $A \cap (B \cap C) = (A \cap B) \cap C$
- **Commutative Properties:** $A \cup B = B \cup A$ and $A \cap B = B \cap A$
- **Identity Property for Union:** $A \cup \varphi = A$
- **Intersection Property of the Empty Set:** $A \cap \varphi = \varphi$
- **Distributive Properties:** $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ similarly for the intersection.

Example : Let A = {0, 2, 4, 6, 8} , B = {0, 1, 2, 3, 4} and C = {0, 3, 6, 9}. What are $A \cup B \cup C$ and $A \cap B \cap C$?

Solution: Set A ∪ B &cup C contains elements that are present in at least one of A, B and C.

$A \cup B \cup C$ = {0, 1, 2, 3, 4, 6, 8, 9}.

Set $A \cap B \cap C$ contains an element that is present in all the sets A, B and C .i.e. { 0 }.

From the definition of a set, it is obvious that:

S u ø: S – ø = S	S n ø = ø	S = S"

De morgan's Law

(S1 U S2)' = S1' ∩ S2'

(S1 ∩ S2)' = S1' U S2'

Operations of Sets:

(1) The union of two sets A and B is such that its elements either belong to A or B. $A \cup B = \{x : x \in A \text{ or } x \in B\}$

Read $A \cup B$ is the set of element x such that either $x \in A$ or $x \in B$.

Properties of union of sets:

1. $A \cap B = B \cap A$.
2. $(A \cap B) \cap C = A \cap (B \cap C)$
3. $A \cap \phi = \phi$
4. $A \cap U = A$, where U is the universal set.
5. $A \cap A = A$
6. If $A \cap B = \phi$, then A and B are disjoint sets.
7. If $A \cap B \neq \phi$, then A and B are overlapping sets.
8. If $A \subset B$, then $A \cap B = A$.
9. If $B \subset A$, then $A \cap B = B$.

(2) The intersection of two sets A and B is the set whose elements belong to A and B both.

$A \cap B = \{x : x \in A \text{ and } x \in B\}$ Read $A \cap B$ as A intersection B.

Definition in symbols: $A - B = \{x : x \in A \text{ and } x \notin B\}$

Example: If $A = \{1, 2, 5, 8, 11\}$,

$B = \{2, 8, 3, 6\}$, then

$A - B = \{1, 5, 11\}$ and $B - A = \{3, 6\}$

Properties of difference of two sets:

1. $A - B = A \cap B'$
2. $A - B = \phi$, if and only if $A \subset B$.
3. $A - B = B - A$, if and only if $A = B$.
4. $A - B = A$, if and only if $A \cap B = \phi$

Table of set theory symbols

Symbol	Symbol Name	Meaning / definition	Example
$\{\ \}$	set	a collection of elements	$A = \{3,7,9,14\}$, $B = \{9,14,28\}$
$\mid$	such that	so that	$A = \{x \mid x \in \mathbb{R}, x<0\}$
$A \cap B$	intersection	objects that belong to set A and set B	$A \cap B = \{9,14\}$
$A \cup B$	union	objects that belong to set A or set B	$A \cup B = \{3,7,9,14,28\}$
$A \subseteq B$	subset	A is a subset of B. set A is included in set B.	$\{9,14,28\} \subseteq \{9,14,28\}$
$A \subset B$	proper subset / strict subset	A is a subset of B, but A is not equal to B.	$\{9,14\} \subset \{9,14,28\}$
$A \not\subset B$	not subset	set A is not a subset of set B	$\{9,66\} \not\subset \{9,14,28\}$
$A \supseteq B$	superset	A is a superset of B. set A includes set B	$\{9,14,28\} \supseteq \{9,14,28\}$
$A \supset B$	proper superset / strict superset	A is a superset of B, but B is not equal to A.	$\{9,14,28\} \supset \{9,14\}$
$A \not\supset B$	not superset	set A is not a superset of set B	$\{9,14,28\} \not\supset \{9,66\}$
2^A	power set	all subsets of A	

Symbol	Symbol Name	Meaning / definition	Example
$\mathcal{P}(A)$	power set	all subsets of A	
$A=B$	equality	both sets have the same members	$A=\{3,9,14\}$, $B=\{3,9,14\}$, $A=B$
A^c	complement	all the objects that do not belong to set A	
A'	complement	all the objects that do not belong to set A	
$A \setminus B$	relative complement	objects that belong to A and not to B	$A = \{3,9,14\}$, $B = \{1,2,3\}$, $A \setminus B = \{9,14\}$
$A-B$	relative complement	objects that belong to A and not to B	$A = \{3,9,14\}$, $B = \{1,2,3\}$, $A - B = \{9,14\}$
$A \triangle B$	symmetric difference	objects that belong to A or B but not to their intersection	$A = \{3,9,14\}$, $B = \{1,2,3\}$, $A \triangle B = \{1,2,9,14\}$
$A \ominus B$	symmetric difference	objects that belong to A or B but not to their intersection	$A = \{3,9,14\}$, $B = \{1,2,3\}$, $A \ominus B = \{1,2,9,14\}$
$a \in A$	element of, belongs to	set membership	$A=\{3,9,14\}$, $3 \in A$
$x \notin A$	not element of	no set membership	$A=\{3,9,14\}$, $1 \notin A$

Symbol	Symbol Name	Meaning / definition	Example
$a \in A$	belongs to	set membership	$A=\{3,9,14\}, 3 \in A$
$x \notin A$	not element of	no set membership	$A=\{3,9,14\}, 1 \notin A$
(a,b)	ordered pair	collection of 2 elements	
$A \times B$	cartesian product	set of all ordered pairs from A and B	
$\|A\|$	cardinality	the number of elements of set A	$A=\{3,9,14\}, \|A\|=3$
$\#A$	cardinality	the number of elements of set A	$A=\{3,9,14\}, \#A=3$
$\aleph_0$	aleph-null	infinite cardinality of natural numbers set	
$\aleph_1$	aleph-one	cardinality of countable ordinal numbers set	
$\emptyset$	empty set	$\emptyset = \{\}$	$A = \emptyset$
$\mathbb{U}$	universal set	set of all possible values	
$\mathbb{N}_0$	natural numbers / whole numbers set (with zero)	$\mathbb{N}_0 = \{0,1,2,3,4,\ldots\}$	$0 \in \mathbb{N}_0$
$\mathbb{N}_1$	natural numbers / whole numbers set (without zero)	$\mathbb{N}_1 = \{1,2,3,4,5,\ldots\}$	$6 \in \mathbb{N}_1$
$\mathbb{Z}$	integer numbers set	$\mathbb{Z} = \{\ldots -3,-$	$-6 \in \mathbb{Z}$

b) Functions and Relations

Function: A function f is a special type of relation in which each first coordinate is associated with one and only one-second coordinates.

Represented as f(x)… element f associates with a given coordinate x.

Example: f(x) =2x + 1.

Have you ever thought that a particular person has particular jobs or functions to do? Consider the functions or roles of postmen. They deliver letters, postcards, telegrams, and invites, etc. What do firemen do? They are responsible for responding to fire accidents. In mathematics also, we can define functions. They are responsible for assigning every single object of one set to that of another.

A function is a relation that maps each element x of a set A with one and only one element y of another set B. In other words, it is a relation between a set of inputs and a set of outputs in which each input is related with a unique output. A function is a rule that relates an input to exactly one output.

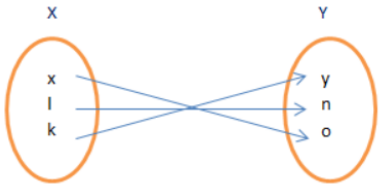
It is a special type of relation. A relation f from a set A to a set B is said to be a function if every element of set A has one and only one image in set B and no two distinct elements of B have the

same mapped first element. A and B are the non-empty sets. The whole set A is the domain and the whole set B is co-domain.

Representation

A function f: X →Y is represented as f(x) = y, where, (x, y) ∈ f and x ∈ X and y ∈ Y.

For any function f, the notation f(x) is read as "f of x" and represents the value of y when x is replaced by the number or expression inside the parenthesis. The element y is the image of x under f and x is the pre-image of y under f.



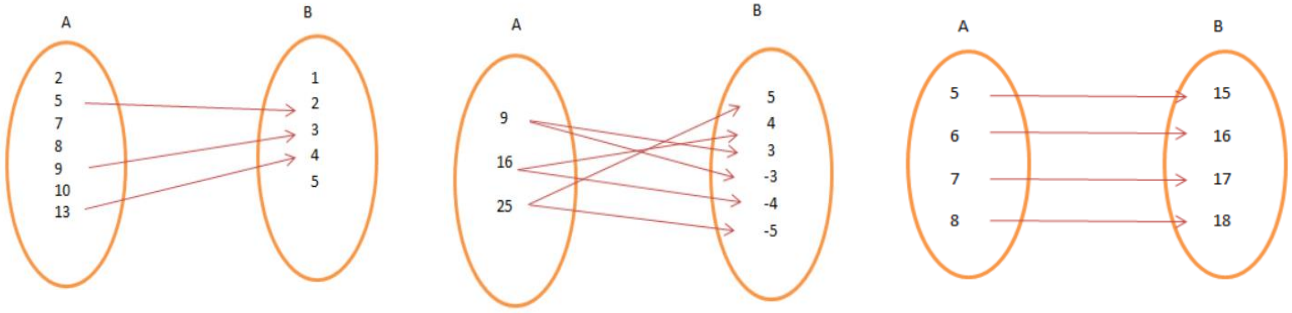
Every element of the set has an image that is unique and distinct. If we notice around, we can find many examples of functions.

If we lift our hand upward, it is a function. Waving our hand freely, it is a function. A walk in a circular track, yes it is a type of function. Now you can think of other examples too! A graph can represent a function. The graph is the set of all pairs of the Cartesian product.

Does this mean that every curve in the world defines a function? No, not every curve drawn is a function. How to find it? Vertical line test. If any curve intercepts a vertical line at more than one point, it is a curve only not a function.

Solved Example for You

Problem: Which of the following is a function?



Solution: Figure 3 is an example of function since every element of A is mapped to a unique element of B and no two distinct elements of B have the same pre-image in A.

A relation is used to describe relationships between members of sets of objects.

Definition. Let *X and Y are set. A relation R from X to Y is simply a* subset of *X x Y. If (a, b)* ϵ *R, we write aRb. If (a, b)* ϵ. *R, we write aRb. If X = Y,* we say R is a **relation** *in X.*

Relation or Binary relation R from set A to B is a subset of AxB which can be defined as aRb ↔ (a,b) € R ↔ R(a,b).

A Binary relation R on a single set A is defined as a subset of AxA. For two distinct sets, A and B with cardinalities m and n, the maximum cardinality of the relation R from A to B is mn.

Domain and Range:

if there are two sets A and B and Relation from A to B is R(a,b), then domain is defined as the set { a | (a,b) € R for some b in B} and Range is defined as the set {b | (a,b) € R for some a in A}.

Types of Relation:

1.Empty Relation: A relation R on a set A is called Empty if the set A is an empty set.

2.Full Relation: A binary relation R on a set A and B is called full if AXB.

3.Reflexive Relation: A relation R on a set A is called reflexive if (a,a) € R holds for every element a € A .i.e. if set A = {a,b} then R = {(a,a), (b,b)} is reflexive relation.

4.Ir-reflexive relation : A relation R on a set A is called reflexive if no (a,a) € R holds for every element a € A. i.e. if set A = {a,b} then R = {(a,b), (b,a)} is ir-reflexive relation.

5.Symmetric Relation: A relation R on a set A is called symmetric if (b,a) € R holds when (a,b) € R. i.e. The relation R={(4,5),(5,4),(6,5),(5,6)} on set A={4,5,6} is symmetric.

6.Anti-Symmetric Relation: A relation R on a set A is called anti-symmetric if (a,b)€ R and (b,a) € R then a = b is called anti-symmetric. i.e. The relation R = {(a,b)→ R|a ≤ b} is anti-symmetric since a ≤ b and b ≤ a implies a = b.

7.Transitive Relation: A relation R on a set A is called transitive if (a,b) € R and (b,c) € R then (a,c) € R for all a,b,c € A. i.e. Relation R={(1,2),(2,3),(1,3)} on set A={1,2,3} is transitive.

8.Equivalence Relation: A relation is an Equivalence Relation if it is reflexive, symmetric, and transitive. i.e. relation R={(1,1),(2,2),(3,3),(1,2),(2,1),(2,3),(3,2),(1,3),(3,1)} on set A={1,2,3} is equivalence relation as it is reflexive, symmetric, and transitive.

9.Asymmetric relation: Asymmetric relation is the opposite of symmetric relation. A relation R on a set A is called asymmetric if no (b,a) € R when (a,b) € R.

Important Points:

1. Symmetric and anti-symmetric relations are not opposite because a relation R can contain both the properties or may not.

2. A relation is asymmetric if and only if it is both anti-symmetric and irreflexive.

3. Number of different relation from a set with n elements to a set with m elements is 2^{mn}

Ex:

if R ={r_1, r_2, r_3.....r_n} and S ={s_1, s_2, s_3.....s_m}

then Cartesian product of R and S is:

R X S = {(r_1, s_1), (r_1, s_2), (r_1, s_3)....., (r_1, s_n),

(r_2, s_1), (r_2, s_2), (r_2, s_3)....., (r_2, s_n),

.....

(r_n, s_1),(r_n, s_2), (r_n, s_3),....., (r_n, s_n)}

This set of ordered pairs contains mn pairs. Now, these pairs can be present in R X S or can be absent. So total number of possible relation = 2^{mn}

4. The number of Reflexive Relations on a set with n elements: $2^{n(n-1)}$.

A relation has ordered pairs (a,b). Now a can be chosen in n ways and same for b. So set of ordered pairs contains n^2 pairs. Now for a reflexive relation, (a,a) must be present in these ordered pairs. And there will be total n pairs of (a,a), so the number of ordered pairs will be n^2-n pairs. So total number of reflexive relations is equal to $2^{n(n-1)}$.

5. Number of Symmetric Relations on a set with n elements : $2^{n(n+1)/2}$.

A relation has ordered pairs (a,b). Now for asymmetric relation, if (a,b) is present in R, then (b,a) must be present in R.

In Matrix form, if a_{12} is present in relation, then a_{21} is also present in relation and As we know reflexive relation is part of symmetric relation.

So from total n^2 pairs, only n(n+1)/2 pairs will be chosen for symmetric relation. So the total number of symmetric relation will be $2^{n(n+1)/2}$.

6. A number of Anti-Symmetric Relations on a set with n elements: $2^n\ 3^{n(n-1)/2}$.

A relation has ordered pairs (a,b). For anti-symmetric relation, if (a,b) and (b,a) is present in relation R, then a = b. (That means a is in relation with itself for any a).
So for (a, a), the total number of ordered pairs = n and total number of relation = 2^n.

if (a,b) and (b,a) both are not present in relation or Either (a,b) or (b,a) is not present in relation. So there are three possibilities and the total number of ordered pairs for this condition is n(n-1)/2. (selecting a pair is same as selecting the two numbers from n without repetition) As we have to find number of ordered pairs where $a \neq b$. it is like opposite of symmetric relation means total number of ordered pairs = (n^2) – symmetric ordered pairs(n(n+1)/2) = n(n-1)/2. So, total number of relation is $3^{n(n-1)/2}$. So total number of anti-symmetric relation is $2^n.3^{n(n-1)/2}$.

7. The number of Asymmetric Relations on a set with n elements: $3^{n(n-1)/2}$.

In Asymmetric Relations, element a cannot be in relation to itself. (i.e. there is no aRa $\forall$ a$\in$A relation.) and then it is the same as Anti-Symmetric Relations.(i.e. you have three choices for pairs (a,b) (b,a)). Therefore there are $3^{n(n-1)/2}$ Asymmetric Relations possible.

8. Ir-reflexive Relations on a set with n elements : $2^{n(n-1)}$.

A relation has ordered pairs (a,b). For Ir-reflexive relation, no (a,a) holds for every element a in R. It is also the opposite of reflexive relation.

Now for an Irreflexive relation, (a, a) must not be present in these ordered pairs means total n pairs of (a, a) are not present in R, So number of ordered pairs will be n^2-n pairs. So total number of reflexive relations is equal to $2^{n(n-1)}$.

9. Reflexive and symmetric Relations on a set with n elements: $2^{n(n-1)/2}$.

A relation has ordered pairs (a,b). Reflexive and symmetric Relations means (a,a) is included in R and (a,b)(b,a) pairs can be included or not. (In Symmetric relation for pair (a,b)(b,a) (considered as a pair). whether it is included in relation or not) So total number of Reflexive and symmetric Relations is $2^{n(n-1)/2}$.

1.4. Mathematical Proof Techniques

An important requirement for reading this text is the ability to follow proofs. In mathematical arguments, we employ the accepted rules of deductive reasoning, and many proofs are simply a sequence of such steps. Two special Proof techniques are used so frequently that it is appropriate to review them briefly. These are proof by induction and proof by contradiction.

Mathematical proof is an argument we give logically to validate a mathematical statement. In order to validate a statement, we consider two things: The number and **Logical operators**.

A statement is either true or false but not both. Logical operators are AND, OR, NOT, If then, and If and only if. Coupled with quantifiers like for all and there exists. We apply operators on the statement to check the correctness of it.

Types of mathematical proofs:

- **Proof by cases –**

In this method, we evaluate every case of the statement to conclude its truthiness.

Example: For every integer x, the integer x(x + 1) is even

Proof: If x is even, hence, x = 2k for some number k. now the statement becomes:

2k(2k + 1)

which is divisible by 2, hence it is even.

If x is odd, hence x = 2k + 1 for some number k, now the statement becomes:

(2k+1)(2k+1+1) = (2k + 1) 2(k + 1)

which is again divisible by 2 and hence in both cases we proved that x(x+1) is even.

- **Proof by Contradiction:**

We assume the negation of the given statement and then proceed to conclude the poof.

Example: Prove that sqrt(2) is irrational

Suppose sqrt(2) is rational.

sqrt(2) = a/b

for some integers a and b with b != 0.

Let us choose integers a and b with sqrt(2) = a/b, such that b is positive and as small as possible. (Well-Ordering Principle)

a^2 = 2b^2

Since a^2 is even, it follows that a is even.

a = 2k for some integer k, so a^2 = 4k^2

b^2 = 2k^2. Since b^2 is even, it follows that b is even.

Since a and b are both even, a/2 and b/2 are integers with b/2 > 0, and sqrt(2) = (a/2)/(b/2), because (a/2)/(b/2) = a/b.

But it contradicts our assumption b is as small as possible. Therefore sqrt(2) cannot be rational.

 Proof by Induction:

The Principle of Mathematical Induction (PMI). Let P(n) be a statement about the positive integer n. If the following are true:

1. P(1),

2. (for all n there exists Z+) P(n) implies P(n + 1),

   then (for all n there exists Z+) P(n).

Example: For every positive integer n,

1 + 2 +⋯+ n = n(n + 1)/ 2

Proof:

Base case: If n = 1,

1 + ⋯ + n = 1

And

n(n + 1)/2 = 11

Inductive Step:

Suppose that for a given n there exists Z+,

1 + 2 +⋯+ n = n(n + 1)/ 2 ---- (i) (inductive hypothesis)

Our goal is to show that:

1 + 2 +⋯+ n + (n + 1) = [n + 1]([n + 1] + 1)/ 2

i.e. 1 + 2 +⋯+ n + (n + 1) = (n + 1)(n + 2) /2

Add n + 1 both sides to equation (i), we get,

1 + 2 +⋯+ n + (n + 1)

= n(n + 1)/ 2 + (n + 1)

= n(n + 1) /2 + 2(n + 1) /2

= (n + 2)(n + 1) /2

 Direct Proof:

when we want to prove a conditional statement p implies q, we assume that p is true, and follow implications to get to show that q is then true.

It is Mostly an application of hypothetical syllogism, $[(p \rightarrow r) \wedge (r \rightarrow q)] \rightarrow (p \rightarrow q)]$ We just have to find the propositions that lead us to q.

Theorem: If m is even and n is odd, then their sum is odd

Proof:

Since m is even, there is an integer j such that m = 2j.

Since n is odd, there is an integer k such that n = 2k+1. Then,

m+n = (2j)+(2k+1) = 2(j+k)+1

Since j+k is an integer, we see that m+n is odd.

Reading Assignment– Practice the above Proof Techniques?

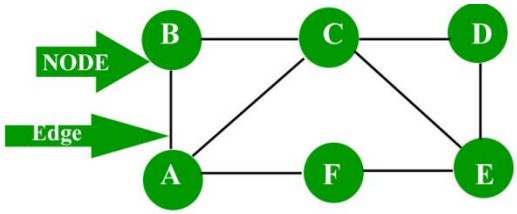
1.5.Graph and Tree

a) Graph: A graph is a data structure that is defined by two components :

1.A **node** or a **vertex, V**.

2.An edge E or **ordered pair is a connection between two nodes u, v** that is identified by unique pair(u, v). The pair (u, v) is ordered because (u, v) is not same as (v, u) in case of directed graph. The edge may have a weight or is set to one in case of un weighted graph.

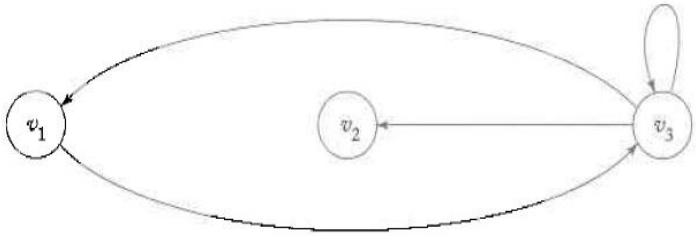
Consider the given below graph,



is a construct consisting of two finite sets, the set V: {v1, v2... vn} of vertices and the set E: {e1, e2... en} of edges.

Each edge consists of two vertices (vi , vj), ei = (vj, vk)

Example: Directed Graph

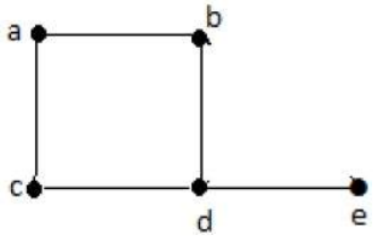


In the domain of mathematics and computer science, *graph theory is the study of graphs that concerns with the relationship among edges and vertices*. It is a popular subject having its applications in computer science, information technology, biosciences, mathematics, and linguistics to name a few. Without further ado, let us start with defining a graph.

What is a Graph?

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices,** and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E),** where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

Applications of Graph Theory

Graph theory has its applications in diverse fields of engineering −

- **Electrical Engineering** − The concepts of graph theory is used extensively in designing circuit connections. The types or organization of connections are named as topologies. Some examples for topologies are star, bridge, series, and parallel topologies.
- **Computer Science** − Graph theory is used for the study of algorithms. For example,
    - Kruskal's Algorithm
    - Prim's Algorithm
    - Dijkstra's Algorithm
- **Computer Network** − The relationships among interconnected computers in the network follows the principles of graph theory.
- **Science** − The molecular structure and chemical structure of a substance, the DNA structure of an organism, etc., are represented by graphs.
- **Linguistics** − The parsing tree of a language and grammar of a language uses graphs.

- **General** − Routes between the cities can be represented using graphs. Depicting hierarchical ordered information such as family tree can be used as a special type of graph called tree.

Graph Terminologies

- **Edges** - as lines with arrows connecting the vertices. Vertices and edges?
- A sequence of edges (vi, vj),(vj, vk),....,(vm, vn) is said to be a walk from vi to vn.
- A **walk:** in which no edge is repeated is said to be a path.
- A **path** is simple if no vertex is repeated.
- A walk from vi to itself with no repeated edge is called a cycle with base vi.
- An edge from a vertex to itself is called a loop

Applications:

Graph is a data structure that is used extensively in our real-life.

1.**Social Network:** Each user is represented as a node and all their activities, suggestion and friend list are represented as an edge between the nodes.

2.**Google Maps:** Various locations are represented as vertices or nodes and the roads are represented as edges and graph theory is used to find the shortest path between two nodes.

3.**Recommendations on e-commerce websites:** The "Recommendations for you" section on various e-commerce websites uses graph theory to recommend items of a similar type to the user's choice.

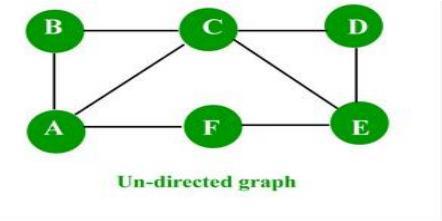
4.Graph theory is also used to **study molecules in chemistry and physics.**

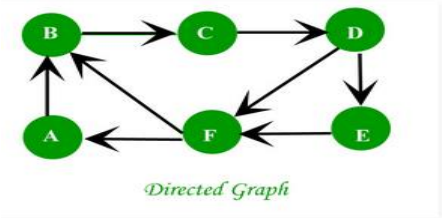
More on graphs:

Characteristics of graphs:

1.**Adjacent node:** A node 'v' is said to be an adjacent node of node 'u' if and only if there exists an edge between 'u' and 'v'.

2.**Degree of a node:** In an undirected graph the number of nodes incident on a node is the degree of the node. In the case of directed graph, **In-degree** of the node is the **number of arriving edges** to a node. **The out-degree** of the node is the **number of departing edges to a node.**





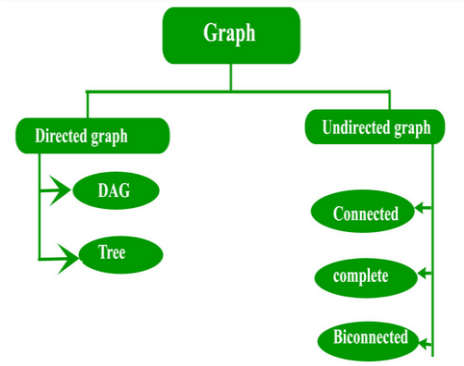
3.**Path**: A path of length 'n' from node 'u' to node 'v' is defined as a **sequence of n+1 nodes.**

P(u,v)=(v0,v1,v2,v3…….vn)

A path is simple if all the nodes are distinct, **exception is source and destination are same.**

4.Isolated node: A node with degree 0 is known as an isolated node. Isolated node can be found by Breadth first search(BFS). It finds its application in **LAN network** in finding whether a **system is connected or not.**

Types of graphs:



1.Directed Graph: A graph in which the direction of the edge is defined to a particular node is a directed graph.

- **Directed Acyclic Graph:** It is a directed graph with no cycle. For a vertex 'v' in DAG there is no directed edge starting and ending with vertex 'v'.

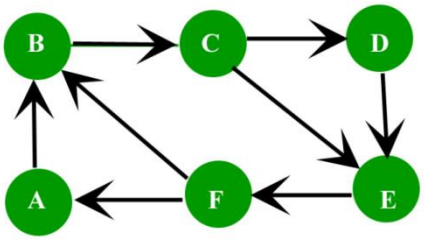
  a) Application: Critical game analysis, expression tree evaluation, game evaluation.

- **Tree**: A tree is just a restricted form of a graph. That is, it is a **DAG with a restriction that a child can have only one parent.**

2.Undirected Graph: A graph in which the direction of the edge is not defined. So if an edge exists between node 'u' and 'v', then there is a path from node 'u' to 'v' and vice versa.

- Connected graph: A graph is connected when there is a **path between every pair of vertices.** In a connected graph, there is no unreachable node.

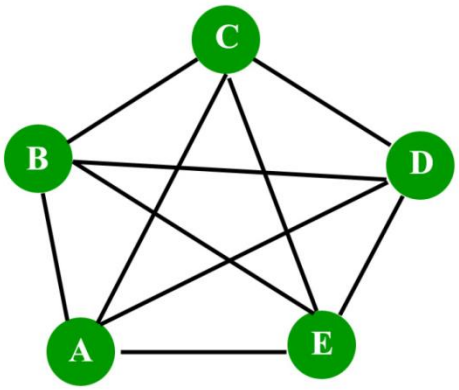
- Complete graph: A graph in which each pair of graph vertices is connected by an edge. In other words, every node ‘u’ is adjacent to every other node ‘v’ in graph ‘G’.A complete graph would have **n(n-1)/2 edges.** See below for proof.
- Bi-connected graph: A connected graph that cannot be broken down into any further pieces by deletion of any vertex. It is a graph with **no articulation point.**



Connected Graph

Proof for a complete graph:

1. Consider a complete graph with n nodes. Each node is connected to other n-1 nodes. Thus it becomes n * (n-1) edges. But this counts each edge twice because this is an undirected graph so divide it by 2.
2. Thus it becomes n(n-1)/2.



Complete Graph

Consider the given graph,

//Omit the repetitive edges

Edges on node A = (A,B),(A,C),(A,E),(A,C).

Edges on node B = (B,C),(B,D),(B,E).

Edges on node C = (C,D),(C,E).

Edges on node D = (D,E).

Edges on node E = EMPTY. See this link: https://en.wikipedia.org/wiki/Graph_theory

Total edges = 4+3+2+1+0=10 edges.

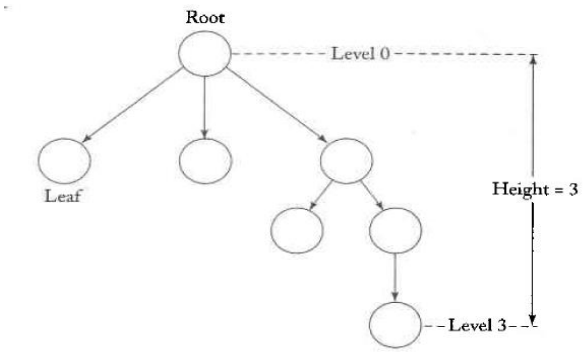
Number of node = 5.

Thus n(n-1)/2=10 edges.

Thus proven.

b) Trees: Are a particular type of graph. A tree is a directed graph that has no cycles and has one distinct vertex called root and has exactly one path from the root to every other vertex

- **Leaves**: are some vertices without outgoing edges.
- **Parent:** if there is an edge from vi to vj, vi is a parent of vj
- **Level:** is the number of edges in the path from the root to the vertex.
- **Height:** of the tree is the largest level number of any vertex.



Reading Assignment: Read about DAG (directed acyclic graph)?

1.6. Core Concepts and Terminologies

- Alphabets
- Language
- Automata
- Strings
- Grammars

Three fundamental ideas are the major themes for this course: **Language, Grammar** and **Automata.** In the course of our study, we will explore our many results about these concepts and relationship to each other. First we must understand the meaning of the terms.

Language: we are all familiar with the notations of natural languages, such as English and French. Still, most of us would probably find it difficult to say exactly what the word 'Language' means. Dictionaries define the term informally as a system suitable for the expression of certain ideas, facts, or concepts including assets of symbols and rules for their manipulation. While this gives us an intuitive idea of what a language is, it is not sufficient as a definition for the study of formal language. We need a precise definition of the term.

1. Alphabet(Σ): Alphabets are a set of symbols, which are always ***finite***.

Examples of alphabets are:

a) {0,1}: alphabet binary

b) {0,1,2,3,4,5,6,7,8,9}: decimal alphabet

c) ASCII, Unicode: machine-text alphabets

d) { }: a legal but not usually interesting alphabet

e) English alphabets

f) Amharic alphabets

2. String: A string is a finite sequence of zero or more symbols from the alphabet. The length of a string abbb is denoted as : |abbb| = 4. A string over the alphabet Σ means a string all of whose symbols are in Σ. The set of all strings of length 2 over the alphabet {a,b} is {aa, ab, ba, bb}. The empty string is written as λ. Like "" in some programming languages. Denoted by |λ| = 0. Don't confuse empty set and empty string:{} ≠ λ

String Operations:

Concatenation: The concatenation of two strings W and Vis the string obtained by appending the symbol of V to the right side of W, that is, if W=a1a2a3…an and V=b1b2…bn the concatenation of W and V is denoted by WV is WV=a1a2…anb1b2…bn.

Length: The length of a string W, denoted by |W|, is the number of symbols in the string.

Empty String: a string with no symbols at all. It will be denoted by λ. The following simple relation | λ |=0, λW=W λ=W holds for all W.

Reversal: The reverse of a string is obtained by writing the symbols in reverse order; if W is a string as shown above, then its reverse WR is WR=an…a2a1.

Substring: any string of consecutive characters in some W is said to be a **substring** of W**.** If W=VU, then the substrings of V and U are said to be a prefix and a suffix of W respectively.

For example: ,if W=abbab,then { λ,a,ab,abb,abbba,abbab} are the set of all prefixes of W,while bab,ab,b arte some of its suffixes.

If W is a string, then W^n stands for the string obtained by repeating W n times. as a special case, we define $W^0 = \lambda$ for all W.

The two common or special sets of strings are:

Σ*: All strings of symbols from Σ concatenation of 0 or more symbols – infinite.

Σ+: Σ* - {λ} –> infinite

Example:

Σ = {0, 1}

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

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

3. Language: A language defined over an alphabet is simply a set of strings over the alphabet. We normally use variable L to stand for a language. A sentence over an alphabet is any string of finite length composed of symbols from the alphabet. Not restricted to finite sets: in fact, finite sets are not usually interesting languages All our alphabets are finite, and all our strings are finite, but most of the languages we're interested in are infinite. A language is defined very generally as a subset of Σ*. A string in a language L will be called a **sentence** of L. Any set of string on an alphabet Σ can be considered a language. For Example, Let Σ={a,b}.then Σ*={ λ,a,b,aa,ab,ba,bb,aaa,aab…}.the set {a,aa,aab} is a Language on Σ. because it has a finite number of sentences, we call it a finite language. The set L= $\{a^n b^n: n \geq 0\}$ is also a language on Σ. The strings aabb and aaaabbbb are in the language L, but abb is not in L. This language is infinite. Most interesting languages are infinite. Language is a set; hence, union, intersection, and difference are defined.

Basic Language Operations

- ***Complement:*** *the complement of a language is defined with respect to Σ*; that is the complement of L is, L' = Σ* - L.*
- ***The reverse of a language:*** *the reverse of a language is the set of all string reversals, that is* $L^R = \{w^R: w \in L\}$

• ***Concatenation:*** *the concatenation of two languages L1 and L2 is the set of all strings obtained by concatenating any elements of L1 with any elements of L2: specifically* *L1L2= {xy: x∈L1, y∈L2}.* We define L^n as L concatenated with itself n times, with special cases $L^0 = \lambda$ and L^1=L for every language L.

• ***Star Closure of L (Kleene Closure)***

$L^* = L^0 \cup L^1 \cup L^2 \ldots$

• ***Positive Closure of L***

$L^+ = L^1 \cup L^2 \cup L^3 \ldots$

Example :if $L=\{a^n b^n : n \geq 0\}$ *then,* $L^2 = \{a^n b^n a^m b^m : n \geq 0, m \geq 0\}$

Notice that n and m in the above are unrelated; the string aabbaaabbb is in L.

The reverse of $L^R = \{b^n a^n, n \geq 0\}$

4. Grammar: Everyday language is imprecise and ambiguous. Informal descriptions in English are often inadequate. Grammar a common and powerful one is introduced. Tell as whether a particular sentence is well formed or not.

In English

‹Sentences› ‹noun-phrase› ‹predicate›

‹noun-phrase› ‹article› ‹noun›

‹predicate› ‹verb›

Example: The dog runs.

Formal Definition: a grammar G is defined as a quadruple (4-tuple).

G=(V, T, S, P) where :

V is a finite set of objects called **variables**

T finite set of object called a **terminal symbol**

S ϵ *V is a special symbol called* ***start variable***

P a finite set of **production**

V and T are non-empty and disjoint.

Production Rule: the heart of grammar, it specifies how a grammar transforms one string into another other.

Define the language associated with the grammar

Described as x→y (*from-to*) … replace x with y and obtain new string. Where x *is an element of* $(V \cup T)^+$ ***and y is in*** $(V \cup T)^*$

Example: w=uxv and z=xyv, then w→z ……. w derives z or z is derived from w. $w_1 \rightarrow w_2 \ldots\ldots \rightarrow w_n$ equals to $w_1 \xrightarrow{*} w_n$, * Represents a useful number of steps are taken to generate w_n. The set of all such terminal strings is the *language* generated by the grammar.

Let G=(V,P,S,T) be a grammar, $L(G)= \{w \in T^*: S \ * \ w\}$

$S \rightarrow w_1 \rightarrow w_2 \ldots\ldots \rightarrow w_n \rightarrow w \ldots$ derivation of the sentence w

$S, w_1, w_2, \ldots, w_n \ldots$ sentential form of the derivation

Example: Consider a grammar G={{S},{a, b},P,T}

P is given: $S \rightarrow aSb \mid S \rightarrow \lambda$

$S \rightarrow aSb \rightarrow aaSbb \rightarrow aabb$

Represented as $S \xrightarrow{*} aabb$, aabb is a sentence in language generated by G

$L(G) = \{a^n b^n : n \geq 0\}$

Proof: S → aSb is recursive?

All sentential forms have the form $w = a^i S b^i$

Apply another production S → aSb on the sentential form resulted in $a^i S b^i \rightarrow a^{i+1} S b^{i+1}$

Since i=1 holds true for all i

apply $S \rightarrow \lambda$ to get a sentence

$S \ * \ a^n S b^n \rightarrow a^n b^n$

G can generate strings of the form $a^n b^n$

Example: Find a Grammar that generates

$L(G)= \{a^n b^{n+1} : n \geq 0\}$

G={{S,A},{a, b},S,P} is the grammar with productions:

$S \rightarrow Ab$

$A \rightarrow aAb$

$A \rightarrow \lambda$

Example: Consider the grammar $G_1= \{\{A,S\},\{a,b\},S,P_1\}$ with P_1 consisting of the production:

$S \rightarrow aAb|\lambda$

$A \rightarrow aAb|\lambda$

The grammar generates the language

$L(G_1)= \{a^n b^n : n \geq 0\}$

5. Automata: is an abstract model of digital computers. Automata include some essential features:

Input file: an automaton read but not change, Input file is divided into cells, each of which holds one symbol. Automata can produce output of some form.

Storage Device: consists of an unlimited number of cells. Automata can read and change the contents of storage cells.

Control Unit: can be any one of a finite number of internal states. The internal state of the control unit is determined by the control unit. Term configuration refers to a particular state of the control unit, input file, and storage unit. The transition of the automata from one configuration to the next is a move. Automata can be **deterministic** or **nondeterministic**

Deterministic automaton is one in which each move is uniquely determined by the current configuration. Possible to determine the future behavior of the automaton

In non-deterministic automata may have several possible moves. Only possible to predicate a set of possible actions. An automaton whose response is limited to a simple "yes" or "no" is called an ***accepter***. Presented with an input string an accepter either accepts the string or rejects. Amore general automaton capable of producing a string of symbols as output is called a ***transducer.***

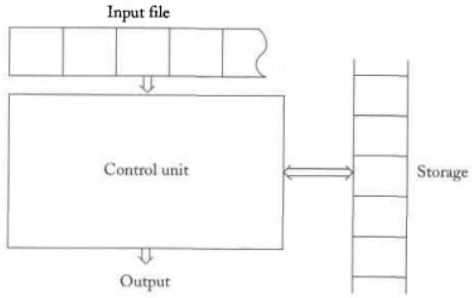


Figure 1: General Model of Automata.

Exercise

1. Let L={ab, aa, baa} which of the following strings are in L^* : abaabaaabaa, aaaabaaaa, baaaaabaaaab, baaaaabaa.

2. Give a simple description of a language generated by the grammar with production

$S \rightarrow aA$ $A \rightarrow bS$ $S \rightarrow \lambda$

3. Find a grammar for $\sum$= {a, b} that generates a set of

a) All strings with exactly one a,

b) All strings with at least one a,

CHAPTER TWO

2. Finite State Automata and Regular Language

Unit Contents:

2.1.Finite State Automata

2.2.Regular Language

2.3.Regular Expression

2.4.Regular Grammar

2.5.Regular Expression and Regular Language

2.6.Regular Grammars and Regular Language

2.7.Regular Grammars and Regular Language and Finite State Automata

2.8.Pumping Lemma

2.1.Finite State Automata

Finite Automata(FA) is the simplest machine to recognize patterns. Finite accepter is characterized by having no temporary storage. Severely limited in its capacity to remember things during the computation.

Types of Automata: **There are two types of finite automata:**

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

A Finite Automata consists of the following :

Q: Finite set of states.

∑: set of Input Symbols.

qo: Initial state.

F: set of Final States.

δ: Transition Function.

Formal specification of machine is

{ Q, ∑, q, F, δ }.

FA is characterized into two types:

A. **Deterministic Finite Accepter (DFA)**

DFA is defined by the 5 tuples:

M=(Q,∑,δ,q0, F)

Where:

Q: a finite set of internal states

∑: a finite symbol called input alphabet

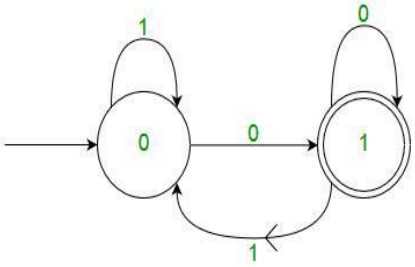
δ: Q × ∑ → Q is a total function called transition function

q0: ε Q is the initial state

F: a subset of Q is a finite set of final states

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

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



One important thing to note is, ***there can be many possible DFAs for a pattern***. A DFA with a minimum number of states is generally **preferred**.

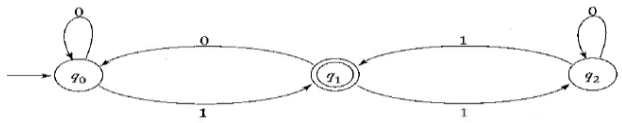
Operates:

Initial state at q0, on the left symbol of the input string. During each move of the automaton, the input mechanism advances one step to right consuming one input symbol. When the end of string is reached the string is accepted if the automaton is in one of its final states, otherwise the string is rejected. The transition from one internal state to the other governed by the transition function.

Example: $\delta(q_0, a) = q_1$, Transition graphs are used to represent finite automata, and the vertices represent the states and the edges represent transitions. The label on the vertices is the names of the state. Labels on the edge are the current value of input symbol.The initial state is identified by an incoming unlabeled arrow not originating at any vertices.

Final state is draw with a double circle

Figure rep. $M=(\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_1\})$



δ is given by

δ(q0, 0)= q0

δ(q0, 1)= q1

δ(q1, 0)= q0

δ(q1, 1)= q2

δ(q2, 0)= q2

δ(q2, 1)= q1

This DFA accepts string 01 and Not accept the sting 00

Extended transition function

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

δ^* is a string rather than a single symbol

δ	0	1
Q0	Q0	Q1
Q1	Q0	Q2
Q2	Q2	Q1

Table 1: Transition table for the above transition function.

B. Nondeterministic Finite Accepter (NFA)

NFA is similar to DFA except following additional features:

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

However, these above features don't add any power to NFA. If we compare both in terms of power, both are equivalent.

Due to the above additional features, NFA has a different transition function, rest is same as DFA.

δ: Transition Function

$\delta: Q \text{ X } (\sum \text{ U } \epsilon) \text{ --> } 2^Q.$

In this case, there are choices of moves for an automaton. Rather than unique moves allow possible moves.

Definition: a Nondeterministic finite accepter of NFA is defined as a 5 Tuple:

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

Where $Q,\sum, q_0$, F are defined as DFA, but $\delta: Q \times (\sum U\{\lambda\}) \rightarrow 2^Q$

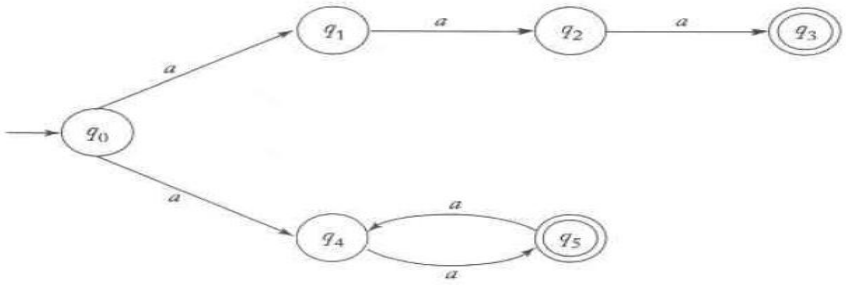
Major differences with DFA:

1.In NFA the range is in the power set of 2^Q. Its value is not a single element of Q but a subset of it.

2.λ is allowed to be the second argument of δ. That means NFA can move without consuming an input symbol.

3.In NFA a set can be defined as $\delta (q_i, a)$ can be empty, meaning there is no transition defined for this situation. Like DFA, NFA can be represented by transition graphs. A string is accepted by the NFA if there are some sequences of possible moves that will put the machine in the final

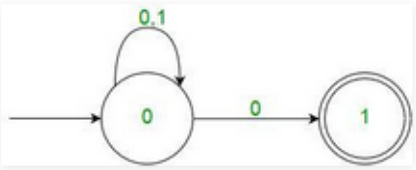
state at the end of the string. A string is rejected if it is not possible to reach the final state. The following transition graph describes nondeterministic finite acceptor since two transitions labeled with a are originated from q_0.



The above transition graph describes a nondeterministic finite accepter since there are two transitions labeled '**a'** out of **q0.**

As you can see in transition function is for any input including null (or ε), NFA can go to any state number of states.

For example, below is an NFA for above problem:



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

Some Important Points:

1. Every DFA is NFA but not vice versa.
2. Both NFA and DFA have the same power and each NFA can be translated into a DFA.
3. There can be multiple final states in both DFA and NFA.
4. NFA is more of a theoretical concept.
5. DFA is used in Lexical Analysis in Compiler.

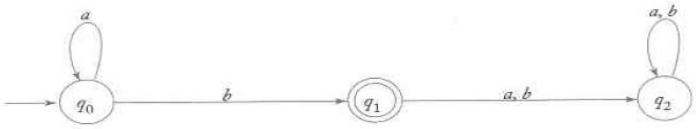
2.2.Language and Finite Automata

Language and DFA: The language is the set of all strings accepted by the automaton

Definition: the language accepted by DFA M= (Q, ∑, δ, q0, F) is the set of all strings ∑ accepted by M.

L(M)={w ε ∑* : δ*(q0,w) ε F}

Example:



The automata accept all strings consists of an arbitrary number of a's followed by b's.

The language accepted by the automaton is L={anb: n≥0}

Q2 is a **trap** state

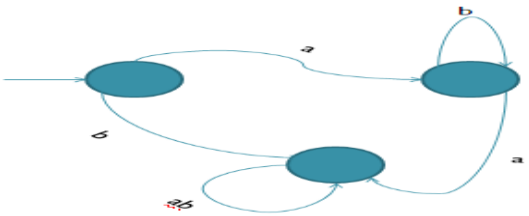
Let M= (Q,∑,δ,q0, F) be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every qi, qjϵ$\sum^+$, δ* (qi,w)=qj if and only if there is in G_M a walk with label w from qi to qj. proof this?

Example 1:

Draw a DFA for the following language

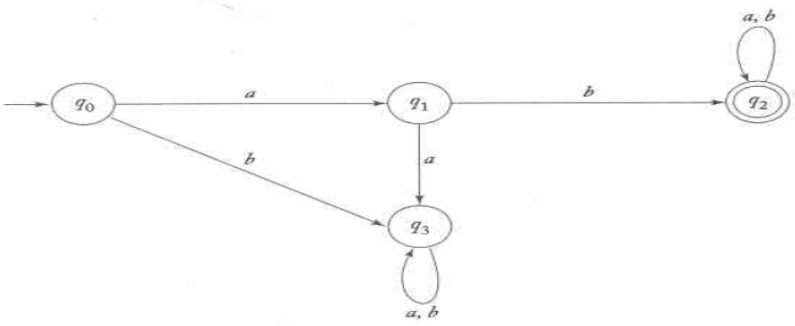
L={ab^n ε∑*: n≥0}

Solution:



Example 2: Find deterministic finite accepter that recognizes the set of all strings on ∑={a,b} starting with prefix ***ab***.

Solution: the only issue here is the first symbols of the string



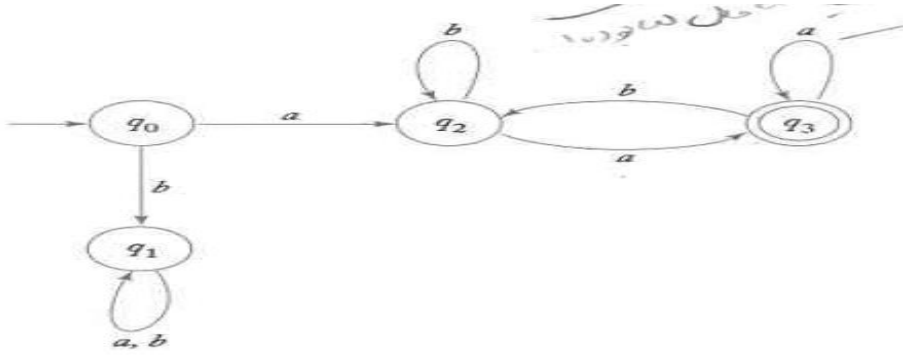
Question: Find a DFA that accepts all the strings on {0, 1}, except those containing the substring 001.

2.3.Regular Language

Definition: A language L is called regular if and only if there exists some deterministic finite accepter M such that: L=M(L)

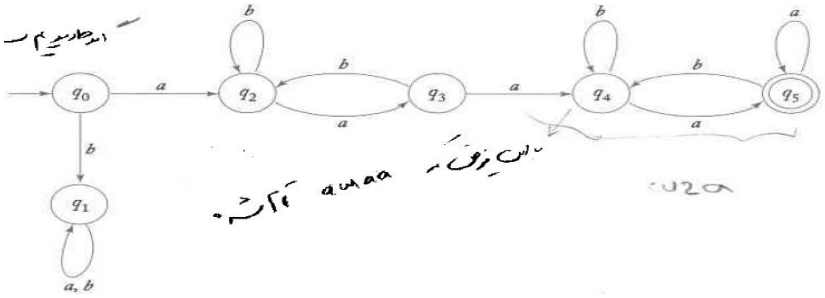
Show that the language, L= {awa: w ε {a, b}*} is regular.

To show the given language is regular, all we have to do is find a DFA for it.



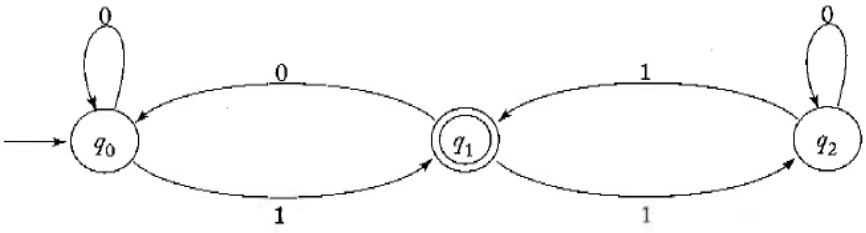
For the previous example show that L^2 is regular language.

L^2={aw1aaw2a: w1, w2 ε {a,b}*}



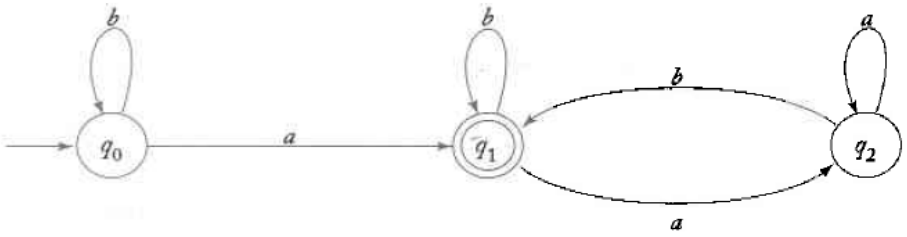
Exercises

Which of the following strings 0001, 01001, 0000110 are accepted the dfa.



2. For ∑ = {a, b}, construct a dfa's that accepts the set consisting of all strings with no more than three a's.

3. Give a set notation description of a language accepted by the automation depicted in the following figure.



2.4.Equivalence of Deterministic and Nondeterministic Finite Accepter

Definition: two finite accepters M1 and M2 are said to be equivalent if L (M1) =L (M2), that is, if they both accept the same language. So any dfa or nfa has many equivalent accepters.

Example: what is the equivalence dfa and nfa for the language L= $\{(10)^n: n\geq 0\}$?

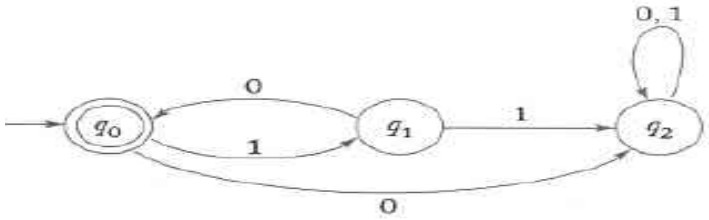


Figure 1.5.Equivalence Dfa for the above language.

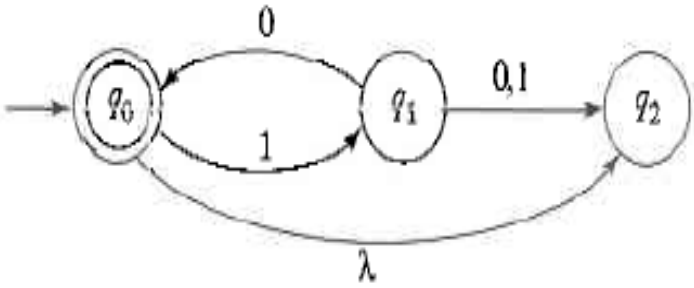


Figure 1.6.eqivalence nfa for the above language.

Conversion from Nfa to Dfa

An NFA can have zero, one or more than one move from a given state on a given input symbol. An NFA can also have NULL moves (moves without input symbol). On the other hand, DFA has one and only one move from a given state on a given input symbol.

Conversion from NFA to DFA

Suppose there is an NFA N < Q, ∑, q0, δ, F > which recognizes a language L. Then the DFA D < Q', ∑, q0, δ', F' > can be constructed for language L as:

Procedures:

Step 1: Initially Q' = ϕ.

Step 2: Add q0 to Q'.

Step 3: For each state in Q', find the possible set of states for each input symbol using transition function of NFA. If this set of states is not in Q', add it to Q'.

Step 4: Final state of DFA will be all states with contain F (final states of NFA)

Example

Consider the following NFA shown in Figure 1:

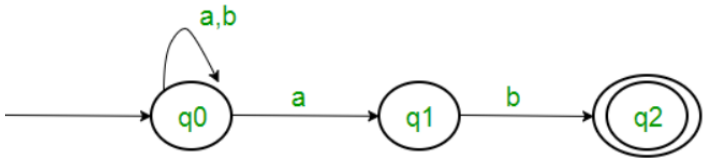


Figure 1

Following are the various parameters for NFA.

Q = { q0, q1, q2 }

∑ = (a, b)

F = { q2 }

δ (Transition Function of NFA)

State	a	b
q0	q0,q1	q0
q1		q2
q2		

Step 1: Q' = ϕ

Step 2: Q' = {q0}

Step 3: For each state in Q', find the states for each input symbol.

Currently, the state in Q' is q0, find moves from q0 on input symbol a and b using transition function of NFA and update the transition table of DFA.

δ' (Transition Function of DFA)

State	a	b
q0	{q0,q1}	q0

Now { q0, q1 } will be considered as a single state. As its entry is not in Q', add it to Q'.

So Q' = { q0, { q0, q1 } }

Now, moves from state { q0, q1 } on different input symbols are not present in transition table of DFA, we will calculate it like:

δ' ({ q0, q1 }, a) = δ (q0, a) ∪ δ (q1, a) = { q0, q1 }

δ' ({ q0, q1 }, b) = δ (q0, b) ∪ δ (q1, b) = { q0, q2 }

Now we will update the transition table of DFA.

δ' (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}

Now { q0, q2 } will be considered as a single state. As its entry is not in Q', add it to Q'.

So Q' = { q0, { q0, q1 }, { q0, q2 } }

Now, moves from state {q0, q2} on different input symbols are not present in transition table of DFA, we will calculate it like:

δ' ({ q0, q2 }, a) = δ (q0, a) ∪ δ (q2, a) = { q0, q1 }

δ' ({ q0, q2 }, b) = δ (q0, b) ∪ δ (q2, b) = { q0 }

Now we will update the transition table of DFA.

δ' (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0

As there is no new state generated, we are done with the conversion. Final state of DFA will be state which has q2 as its component i.e., { q0, q2 }

Following are the various parameters for DFA.

Q' = { q0, { q0, q1 }, { q0, q2 } }

∑ = (a, b)

F = { { q0, q2 } } and transition function δ' as shown above. The final DFA for above NFA has been shown in Figure 2.

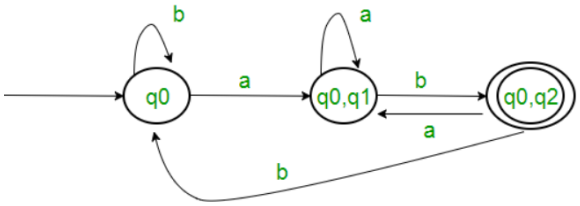


Figure 2

Note : Sometimes, it is not easy to convert regular expression to DFA. First you can convert regular expression to NFA and then NFA to DFA.

Question : The number of states in the minimal deterministic finite automaton corresponding to the regular expression (0 + 1)* (10) is_____ ?

Reduction of the Number of States in Finite Automata

DFA minimization stands for converting a given DFA to its equivalent DFA with minimum number of states.

2.5.Minimization of DFA

Suppose there is a DFA D < Q, Σ, q0, δ, F > which recognizes a language L. Then the minimized DFA D < Q', Σ, q0, δ', F' > can be constructed for language L as:

Step 1: We will divide Q (set of states) into two sets. One set will contain all final states and other set will contain non-final states. This partition is called P_0.

Step 2: Initialize k = 1

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

Step 4: Stop when $P_k = P_{k-1}$ (No change in partition)

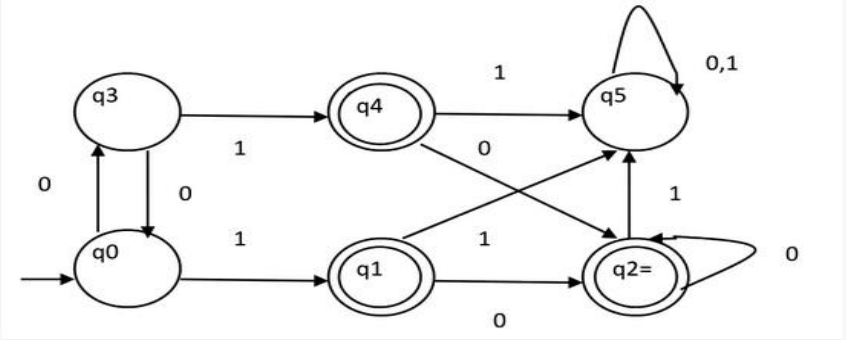
Step 5: All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of sets in P_k.

How to find whether two states in partition P_k are distinguishable ?

Two states (qi, qj) are distinguishable in partition P_k if for any input symbol a, δ (qi, a) and δ (qj, a) are in different sets in partition P_{k-1}.

Example

Consider the following DFA shown in figure.



2.6.Regular Expressions: According to our definition, a language is regular if there is a finite accepter for it. Therefore every language can be described by some dfa or some nfa. Such a description can be very useful, for example if we want to show the logic by which we decide if a given string is in a certain language. But in any instances, we need more concise ways of describing regular languages. In this chapter, we look at other ways of representing regular language. These representations have important practical applications, a matter that is touched on in some of the examples and exercises.

One way of describing regular languages is via the notation of regular expressions. Involves the combination of strings of symbols from some alphabet Σ, parenthesis and, the operator +, * and dot(.).

A **Regular Expression** can be recursively defined as follows −

- **ε** is a Regular Expression indicates the language containing an empty string. **(L (ε) = {ε})**
- **φ** is a Regular Expression denoting an empty language. **(L (φ) = { })**
- **x** is a Regular Expression where **L = {x}**
- If **X** is a Regular Expression denoting the language **L(X)** and **Y** is a Regular Expression denoting the language **L(Y)**, then
    - **X + Y** is a Regular Expression corresponding to the language **L(X) ∪ L(Y)** where **L(X+Y) = L(X) ∪ L(Y)**.
    - **X . Y** is a Regular Expression corresponding to the language **L(X) . L(Y)** where **L(X.Y) = L(X) . L(Y)**
    - **R*** is a Regular Expression corresponding to the language **L(R*)**where **L(R*) = (L(R))***
- If we apply any of the rules several times from 1 to 5, they are Regular Expressions.

Some RE Examples

Regular Expressions	Regular Set

(0 + 10*)	L = { 0, 1, 10, 100, 1000, 10000, … }
(0*10*)	L = {1, 01, 10, 010, 0010, …}
(0 + ε)(1 + ε)	L = {ε, 0, 1, 01}
(a+b)*	Set of strings of a's and b's of any length including the null string. So L = { ε, a, b, aa , ab , bb , ba, aaa…….}
(a+b)*abb	Set of strings of a's and b's ending with the string abb. So L = {abb, aabb, babb, aaabb, ababb, ……..}
(11)*	Set consisting of even number of 1's including empty string, So L= {ε, 11, 1111, 111111, …….}
(aa)*(bb)*b	Set of strings consisting of even number of a's followed by odd number of b's , so L = {b, aab, aabbb, aabbbbb, aaaab, aaaabbb, ……..}
(aa + ab + ba + bb)*	String of a's and b's of even length can be obtained by concatenating any combination of the strings aa, ab, ba and bb including null, so L = {aa, ab, ba, bb, aaab, aaba, ……..}

Regular Sets

Any set that represents the value of the Regular Expression is called a **Regular Set.**

Properties of Regular Sets

Property 1. *The union of two regular set is regular.*

Proof –

Let us take two regular expressions

$RE_1 = a(aa)^*$ and $RE_2 = (aa)^*$

So, L_1 = {a, aaa, aaaaa,.....} (Strings of odd length excluding Null)

and L_2 ={ ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 \cup L_2$ = { ε, a, aa, aaa, aaaa, aaaaa, aaaaaa,.....}

(Strings of all possible lengths including Null)

RE ($L_1 \cup L_2$) = a* (which is a regular expression itself)

Hence, proved.

Property 2. *The intersection of two regular set is regular.*

Proof –

Let us take two regular expressions

$RE_1 = a(a^*)$ and $RE_2 = (aa)^*$

So, L_1 = { a,aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 \cap L_2$ = { aa, aaaa, aaaaaa,.....} (Strings of even length excluding Null)

RE ($L_1 \cap L_2$) = aa(aa)* which is a regular expression itself.

Hence, proved.

Property 3. *The complement of a regular set is regular.*

Proof −

Let us take a regular expression −

RE = (aa)*

So, L = {ε, aa, aaaa, aaaaaa, ...} (Strings of even length including Null)

Complement of **L** is all the strings that is not in **L**.

So, L' = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

RE (L') = a(aa)* which is a regular expression itself.

Hence, proved.

Property 4. *The difference of two regular set is regular.*

Proof −

Let us take two regular expressions −

RE_1 = a (a*) and RE_2 = (aa)*

So, L_1 = {a, aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 – L_2$ = {a, aaa, aaaaa, aaaaaaa, ...}

(Strings of all odd lengths excluding Null)

RE ($L_1 – L_2$) = a (aa)* which is a regular expression.

Hence, proved.

Property 5. *The reversal of a regular set is regular.*

Proof −

We have to prove $\mathbf{L^R}$ is also regular if **L** is a regular set.

Let, L = {01, 10, 11, 10}

RE (L) = 01 + 10 + 11 + 10

L^R = {10, 01, 11, 01}

RE (L^R) = 01 + 10 + 11 + 10 which is regular

Hence, proved.

Property 6. *The closure of a regular set is regular.*

Proof −

If L = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

i.e., RE (L) = a (aa)*

L* = {a, aa, aaa, aaaa , aaaaa,……} (Strings of all lengths excluding Null)

RE (L*) = a (a)*

Hence, proved.

Property 7. *The concatenation of two regular sets is regular.*

Proof −

Let RE_1 = (0+1)*0 and RE_2 = 01(0+1)*

Here, L_1 = {0, 00, 10, 000, 010, ...} (Set of strings ending in 0)

and L_2 = {01, 010,011,.....} (Set of strings beginning with 01)

Then, $L_1 L_2$ = {001,0010,0011,0001,00010,00011,1001,10010,.....}

Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)*001(0 + 1)*

Hence, proved.

Identities Related to Regular Expressions

Given R, P, L, Q as regular expressions, the following identities hold −

- ∅* = ε
- ε* = ε
- RR* = R*R
- R*R* = R*
- (R*)* = R*
- RR* = R*R
- (PQ)*P =P(QP)*
- (a+b)* = (a*b*)* = (a*+b*)* = (a+b*)* = a*(ba*)*
- R + ∅ = ∅ + R = R (The identity for union)
- R ε = ε R = R (The identity for concatenation)
- ∅ L = L ∅ = ∅ (The annihilator for concatenation)
- R + R = R (Idempotent law)
- L (M + N) = LM + LN (Left distributive law)
- (M + N) L = ML + NL (Right distributive law)
- ε + RR* = ε + R*R = R*

2.7. Arden's Theorem

In order to find out a regular expression of a Finite Automaton, we use Arden's Theorem along with the properties of regular expressions.

Statement −

Let **P** and **Q** be two regular expressions.

If **P** does not contain null string, then **R = Q + RP** has a unique solution that is **R = QP***

Proof −

R = Q + (Q + RP)P [After putting the value R = Q + RP]

= Q + QP + RPP

When we put the value of **R** recursively again and again, we get the following equation −

$R = Q + QP + QP^2 + QP^3$…..

$R = Q (\varepsilon + P + P^2 + P^3 + \ldots.)$

R = QP* [As P* represents (ε + P + P2 + P3 + ….)]

Hence, proved.

Assumptions for Applying Arden's Theorem

- The transition diagram must not have NULL transitions
- It must have only one initial state

Method

Step 1 − Create equations as the following form for all the states of the DFA having n states with initial state q_1.

$q_1 = q_1R_{11} + q_2R_{21} + \ldots + q_nR_{n1} + \varepsilon$

$q_2 = q_1R_{12} + q_2R_{22} + \ldots + q_nR_{n2}$

……

……

……

……

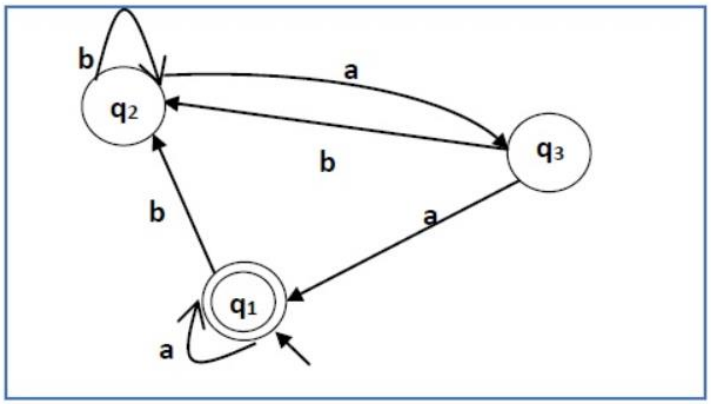
$q_n = q_1R_{1n} + q_2R_{2n} + \ldots + q_nR_{nn}$

$\mathbf{R_{ij}}$ represents the set of labels of edges from $\mathbf{q_i}$ to $\mathbf{q_j}$, if no such edge exists, then $\mathbf{R_{ij} = \emptyset}$

Step 2 − Solve these equations to get the equation for the final state in terms of $\mathbf{R_{ij}}$

Problem

Construct a regular expression corresponding to the automata given below −



Solution −

Here the initial state and final state is $\mathbf{q_1}$.

The equations for the three states q1, q2, and q3 are as follows −

$q_1 = q_1a + q_3a + \varepsilon$ (ε move is because q1 is the initial state0

$q_2 = q_1b + q_2b + q_3b$

$q_3 = q_2a$

Now, we will solve these three equations −

$q_2 = q_1b + q_2b + q_3b$

$= q_1b + q_2b + (q_2a)b$ (Substituting value of q_3)

$= q_1b + q_2(b + ab)$

$= q_1b\ (b + ab)^*$ (Applying Arden's Theorem)

$q_1 = q_1a + q_3a + \varepsilon$

$= q_1a + q_2aa + \varepsilon$ (Substituting value of q_3)

$= q_1a + q_1b(b + ab^*)aa + \varepsilon$ (Substituting value of q_2)

$= q_1(a + b(b + ab)^*aa) + \varepsilon$

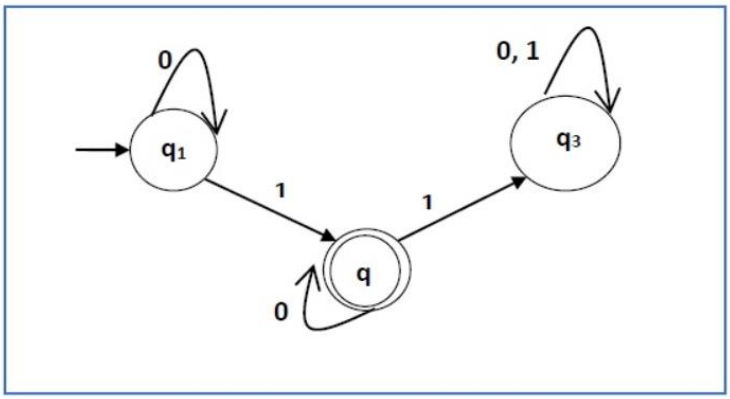
$= \varepsilon\ (a+ b(b + ab)^*aa)^*$

$= (a + b(b + ab)^*aa)^*$

Hence, the regular expression is $(a + b(b + ab)^*aa)^*$.

Problem

Construct a regular expression corresponding to the automata given below −



Solution −

Here the initial state is q_1 and the final state is q_2

Now we write down the equations −

$q_1 = q_10 + \varepsilon$

$q_2 = q_11 + q_20$

$q_3 = q_21 + q_30 + q_31$

Now, we will solve these three equations −

$q_1 = \varepsilon 0^*$ [As, $\varepsilon R = R$]

So, $q_1 = 0^*$

$q_2 = 0^*1 + q_20$

So, $q_2 = 0^*1(0)^*$ [By Arden's theorem]

Hence, the regular expression is 0^*10^*.

Example: {a} can be represented as a regular expression. In RE we use:

+	Union
.	concatenation
*	star-closure

Example:

(a + b .c)* stands for the star-closure of {a}, λ ,{bc} , viz, { λ, a, bc, abc,bca, aa,bcbc,... }

Definition: Let Σ be a given alphabet. Then

1. Ø, ʎ, and a ϵ Σ are all regular expressions. These are called primitive regular expression.
2. If r_1 and r_2 are regular expressions, so are $r_1 + r_2$, r $r_1 . r_2$, r_1^*, and (r_1).
3. A string is a regular expression if and only if it can be derived from the primitive regular expressions by a finite number of applications of the rules in (2).

Example: Σ = {a, b, c} then (a + b. c)*(c + ʎ) is a regular expression but not (a + b.)

2.2.Languages associated with regular expression

If r is a regular expression we will let L (r) denote the language associated with r.

Definition: the language L(r) denoted by any regular expression r is defined by the following rules.

1. Ø is a regular expression denoting the empty set,
2. ʎ is a regular expression denoting{ʎ},
3. For every a ϵ Σ, a is a regular expression denoting {a}.

If r_1 and r_2 are regular expressions, then

1.$L(r_1 + r_2) = L(r_1) \cup L(r_2)$

2.$L(r_1 . r_2) = L(r_1) L(r_2)$

3.$L((r_1)) = L(r_1)$

4.$L(r_1^*) = (L(r_1))^*$

4-7 reduce L(r) to simpler components recursively, but 1-3 are the termination conditions.

Example: exhibit the language L (a*.(a+b)) in set notation

L (a*.(a+b)) = L(a*)L(a+b)

= (L (a))*(L {a} U L {b})

= {ʎ, a,aa, aaa, …}{a,b}

= {ʎ, a,aa, aaa, b,ab,aab,aaab, ….}

For ambiguous expression use

Example: for Σ = {a, b}, the expression r= (a + b)*(a + bb) is regular it denotes the language L(r) = { a, bb, aa, abb, ba, bbb, … }

(a + b)* represents any string either a's and b's

(a + bb) represents either an a or double b ,

Therefore, L(r) is the set of all strings on {a, b}, terminated by either an a or bb

Example: the expression r= (aa)*(bb)*b denotes the set of all strings with even number of a's and odd number of b's , that is

$L=\{a^{2n}b^{2m+1}: n\geq 0, m\geq 0\}$

Example: For Σ = {0, 1}, give a regular expression such that:

L(r) = {w ε Σ* : w has atleas to one pair of consequtive zeros }

L(r) must contains consecutive zeros somewhere in w, but what comes after and before is arbitrary

An arbitrary {0,1} is denoted as (0+1)*, we can arrive the solution r=(0+1)*00(0+1)*

Example: Find a regular expression for the language L={w ε {0,1}*: w has no pair of consecutive zeros }

When 0 occur it must be followed immediately by 1... 1…1011…. Implies (1*011*)*

r= (1+01)*(0+ λ)

Generally, there are an unlimited number of regular expressions for any given language.

We say the two regular expressions are equivalent if they denote the same language.

Construction of an FA from an RE

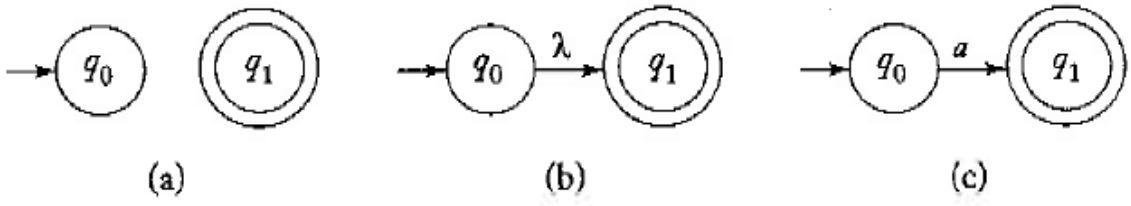
Refer: https://www.tutorialspoint.com/automata_theory/constructing_fa_from_re.htm

2.8.Connection between Regular Expressions and regular languages

For every regular language there is regular expression and for every regular expression there is a regular language. if r is a regular expression, then L (r) is a regular language. Previous definition says that a language is regular if it is accepted by some dfa. Because of the equivalence of nfa's and dfa's, a language is also regular if it is accepted by some nfa. Thus, if we have a regular expression r, we can draw an nfa that accept L(r). The construction of nfa relies of the recursive definition of L(r)

Theorem: Let r be a regular expression. Then there exist nfa that accepts L(r). Consequently, L(r) is regular language.

Proof: we begin with the automaton that accepts the language for simple regular expressions: Ø, λ, a ε Σ. These are shown in (a), (b) and (c).



(a) (b) (c)

(a).nfa accepts Ø.

(b).nfa accepts λ.

(c).nfa accepts a.

A grammar is a set of production rules which are used to generate strings of a language. In this article, we have discussed how to find the language generated by a grammar and vice versa as well.

Language generated by a grammar

Given a grammar G, its corresponding language L(G) represents the set of all strings generated from G. Consider the following grammar,

```
G: S-> aSb|ε
```

In this grammar, using S-> ε, we can generate ε. Therefore, ε is part of L(G). Similarly, using S=>aSb=>ab, ab is generated. Similarly, aabb can also be generated.
Therefore,
$L(G) = \{a^n b^n, n>=0\}$
In language L(G) discussed above, the condition n = 0 is taken to accept ε.
Key Points –

- For a given grammar G, its corresponding language L(G) is unique.
- The language L(G) corresponding to grammar G must contain all strings which can be generated from G.
- The language L(G) corresponding to grammar G must not contain any string which cannot be generated from G.

2.9.Regular grammar

A third ways of describing a regular language is by means of certain simple grammars. Grammars are often an alternative ways of specifying languages. Whenever we define a languages family through an automaton or in some other ways, we are interested in knowing what kind of grammars we can associate with the family. First we look at grammars that generate regular languages.

Grammar is an alternative way of describing regular languages

Grammars can be:

Right and left linear grammars

A grammar G = (V,T,S,P) is said to be right-linear if all productions are of the form:

$A \rightarrow xB$,

$A \rightarrow x$….. Where A and B ϵ V and x ϵ T*

A grammar is said to be left-linear if all production are of the form

$A \rightarrow Bx$,

$A \rightarrow x$

A regular grammar is either right or left

Example: the grammar $G_1=(\{S\},\{a,b\},S,P_1)$ with production $S \rightarrow abS|a$ is right linear

The grammar $G_2=(\{S_1,S_2,S_3\}, \{a,b\},S, P_2)$with production

$S \rightarrow abS_1,|$

$S_1 \rightarrow abS_1|S_2$,

$S_2 \rightarrow a$

Is left-linear. Both G_1 and G_2 are regular grammars

We can say that

$L(G_1)$ is a language denoted by a regular expression r=(ab)*a

$L(G_2)$ is a regular language L(aab(ab)*)

Example: the Grammar G=({S,A,B},{a,b},S,P) with production

$S \rightarrow A$,

$A \rightarrow aB|\lambda$,

$B \rightarrow Ab$, is not a regular grammar, but the grammar is linear grammar since at most one variable is occur at the right said of any production. Regular grammar is always linear but linear grammar is not always regular.

2.10. Right Linear Grammar Generate Regular Language

A language generated by a right-linear grammar is always regular

Construct an nfa which mimics the derivation of right linear grammar

For instance $D \rightarrow dE$, means the nfa transform from D to E by taking d as input symbol

Theorem: Let G= (V, T, S, P) be a right linear grammar. Then L (G) is a regular language.

Proof: we assume that $V= \{V_0, V_1 \ldots\}$, $S= V_0$ and that we have a production of the form $V_0 \rightarrow v_1V_i$, $V_i \rightarrow v2V_j \ldots$ or $V_n \rightarrow v_l$. If w is a string in L (G). The derivation must have the form

$V_0 \rightarrow v_1V_i$

$\rightarrow v_1v_2V_j$

$\rightarrow v_1v_2 \ldots\ldots v_kV_n$

$\rightarrow v_1v_2 \ldots\ldots v_k v_l = w$

2.11. Properties Of Regular Languages

1. Closure properties of RL

Closure under simple set operation

Recall a closure property is a statement that a certain operation on languages, when applied to languages in a class (e.g., the regular languages), produces a result that is also in that class.

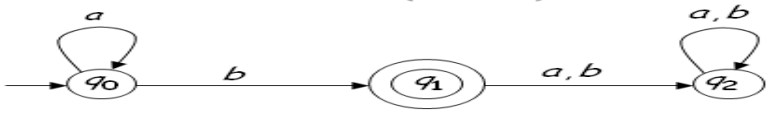
Theorem 4.1. If L1 and L2 are regular language, then so are L1U L2, L1∩ L2, L1L2, L1 and L1*.

Thus, the family of regular language is closed under

- Union. If L an M is RL, then S & R r RE for M and S.. S + R is RL
- Intersection
- Concatenation
- Complement
- Star- closure

Example:

$$L = L(a^* b)$$



$$\overline{L} = L(a^* + a^* b(a+b)(a+b)^*)$$

For regular languages L_1 and L_2 the intersection $L_1 \cap L_2$ is regular

L_1 , L_2 regular

$\overline{L_1}$, $\overline{L_2}$ regular

$\overline{L_1} \cup \overline{L_2}$ regular

$\overline{\overline{L_1} \cup \overline{L_2}}$ regular

$L_1 \cap L_2$ regular

Demorgan's law representation is regular.

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Example: Show that the family of regular language is closed under difference.

If L1 and L2 regular language so is L1 - L2

Proof: we know that L1 - L2 = $L_1 \cap \overline{L_2}$.

We know that if L2 is regular $\overline{L_2}$ also regular, because of the closure property of regular language under intersection $L_1 \cap \overline{L_2}$. is also regular.

Theorem: The family of regular languages is close under reversal.

Proof: Suppose L is regular language, and then construct an nfa with a single final state.

Make the initial vertex the final vertex and the final vertex the initial vertex and reverse the direction of all the edges.

The modified nfa accepts w^R if and only if w is accepted by the original nfa.

Closure under other operations

Definition 4.1. Suppose Σ and Γ. Then a function

h: $\Sigma \rightarrow \Gamma^*$ is called homomorphism.

A homomorphism on an alphabet is a function that gives a string for each symbol in that alphabet.

If $w = a_1a_2\ldots..a_n$, then $h(w)= h(a_1)\ h(a_2)\ldots\ h(a_n)$

lf L is a, language on Σ, then its homomorphic image is defined as:

h(L)={ h(w): w □ L }

Example: if Σ={a, b} and Γ={a, b, c} and h define by

h(a)= ab h(b)= bbc then h(aba) is = abbbcab

The homomorphic image of L={aa,aba} is the language h(L)={abab, abbbcab}

Take Σ={a, b} and Γ={b, c, d}. Define h by

h(a)= dbcc h(b) = bdc

If L is a regular language denoted by

r= (a + b*)(aa)* then h(L) can be represented as

r=(dbcc + (bdc)*)(dbccdbcc)*

Theorem: If L is a regular language then its homomorphic image h (L) is also regular.

The family of regular languages is therefore closed under an arbitrary homomorphism.

Proof: reading assignment

Definition: Let L_1 and L_2 be languages on the same alphabet. Then the **right quotient** of L_1 with L_2 is defined as:

$L_1/L_2 = \{x:xy\ \varepsilon L_1$ for some y $\varepsilon L_2\}$

To form right quotient of L_1 with L_2 we can talk all strings in L_1 that have a suffixes belongs to L_2.

Every such string after the removal of this suffixes, belongs to L_1/L_2 .

Example: if $L_1 = \{a^nb^m : n \geq 0, m \geq 0\}$ U {ba} and $L2 = \{b^m : m \geq 1\}$ then

$L_1/L_2 = \{a^nb^m : n \geq 0, m \geq 0\}$

The strings in L2 consist of one or more b's.

Therefore, we arrive at the answer by removing one or more b's from those strings in L_1 that terminates with at least one b's as suffix.

Theorem: If L1 and L2 are regular languages, the L_1/L_2 is also regular.

Proof: Let L1= L(M) where (Q,Σ,δ,q_0,F) is a dfa.

Construct another dfa

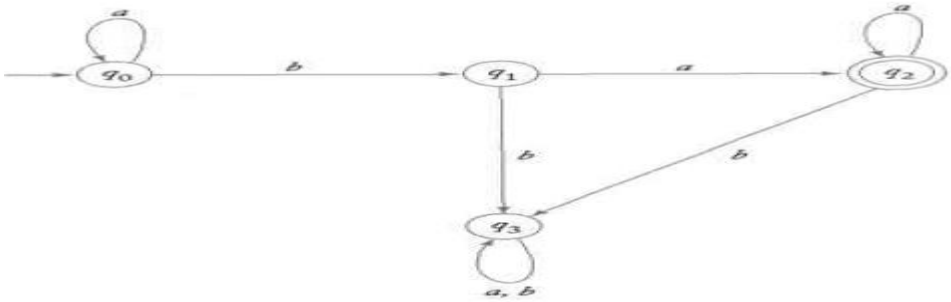
Possible by constructing a dfa for L_1/L_2

Example: Find L_1/L_2 for

L_1= L(a*baa*)

L_2= L(ab*)

Find a dfa that accepts L_1



Construct another dfa as follow. For each qi in Q determine if there exists a y ε L2 such that, δ*(qi, y) = qf ε F

This can be looking at the dfa's Mi = (Q,Σ,δ, qi, F). The automaton Mi is M with the initial state q0 replaced by qi.

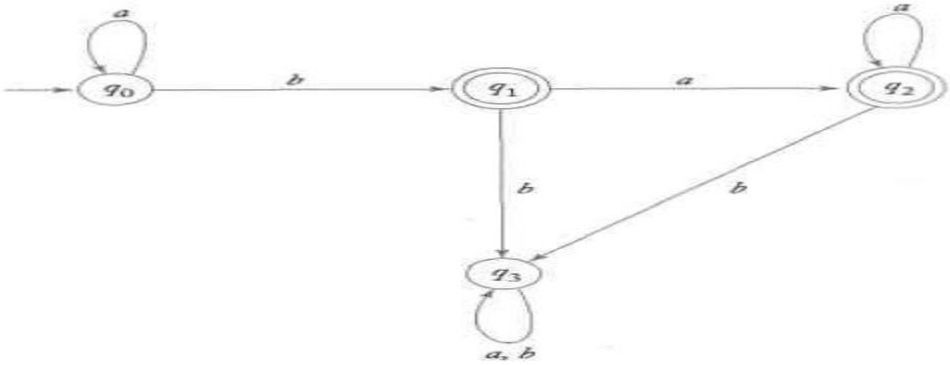
Then, determine whether there exist a y in L(Mi) is also in L2.

Therefore, use intersection for the construction of dfa.

Find a transition graph for L2 n L(Mi). lf there is any path between its initial vertex and any final vertex, L2 n L(Mi) is not empty.

In this case add qi in F..

Repeat this for every qi ε Q, we determine F.. and construct M..



2.12. Identifying Non Regular Languages

Brainstorming: How can we prove that a language is not regular?

Prove that there is no DFA that accepts L.

Problem: this is not easy to prove.

Solution: the Pumping Lemma!!!

A Pumping Lemma

The pumping lemma helps us to prove a language not to be regular.

Used only for infinite languages, finite languages cannot be pumped.

Theorem 4.8. Let L be an infinite regular language. Then there exist some positive integer m such that any w ε L with |w| ≥ m can be decomposed as

w= xyz |xy|≤ m |y|≥ 1 such that $w_i = xy^iz$ is also in L for i = 0, 1, 2 …

Using pumping lemma show that the L= $\{a^nb^n: n\geq 0\}$ is not regular.

So/n: Assume that L is regular. We can choose m=n for unknown value of m.

In this case the substring y consists of entirely a's

Suppose |y|=k. The substring obtained by using i=0 in $w_i = xy^iz$ is

W0= $a^{m-k}b^m$ not in L.

This contradicts the pumping lemma and it is not regular.

Theorem

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

|w| ≥ c

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

- |y| > 0
- |xy| ≤ c
- For all k ≥ 0, the string xy^kz is also in L.

Applications of Pumping Lemma

Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular.

- If **L** is regular, it satisfies Pumping Lemma.
- If **L** does not satisfy Pumping Lemma, it is non-regular.

Method to prove that a language L is not regular

- At first, we have to assume that **L** is regular.
- So, the pumping lemma should hold for **L**.
- Use the pumping lemma to obtain a contradiction −
    - Select **w** such that **|w| ≥ c**
    - Select **y** such that **|y| ≥ 1**
    - Select **x** such that **|xy| ≤ c**
    - Assign the remaining string to **z.**
    - Select **k** such that the resulting string is not in **L.**

Hence L is not regular.

Problem

Prove that **L = {a^ib^i | i ≥ 0}** is not regular.

Solution −

- At first, we assume that **L** is regular and n is the number of states.
- Let w = a^nb^n. Thus |w| = 2n ≥ n.
- By pumping lemma, let w = xyz, where |xy| ≤ n.
- Let x = a^p, y = a^q, and z = a^rb^n, where p + q + r = n, p ≠ 0, q ≠ 0, r ≠ 0. Thus |y| ≠ 0.
- Let k = 2. Then $xy^2z = a^pa^{2q}a^rb^n$.
- Number of a's = (p + 2q + r) = (p + q + r) + q = n + q
- Hence, $xy^2z = a^{n+q}b^n$. Since q ≠ 0, xy^2z is not of the form a^nb^n.
- Thus, xy^2z is not in L. Hence L is not regular.

Proof:

Assume that A is regular

Pumping length=p

$S=a^nb^n$ =>S=aaaaaaabbbbbbb, n=7

S(X,Y,Z)

Case 1: The 'Y' is in the 'a' part | $xy^iz=>xy^2z$ No

aaaaaaabbbbbbb(X,Y,Z) | aa aaaa aaaa a bbbbbbb, 11≠7

Case 2: The 'Y' is in the 'b' part. | $xy^iz=>xy^2z$ No

aaaaaaabbbbbbb(X,Y,Z) | aaaaaaa bb bbbb bbbb b, 7≠11

Case 3:The 'Y' is in the 'a' and 'b' part. | xyiz=>xy2z No

aaaaaaabbbbbbb(X,Y,Z) | aaaa aabbaabb bbbbb, this violates the form a^nb^n.

|xy|<=p, p=7

Summary:

>>Pumping Lemma is used to prove that a language is NOT Regular.

>>It cannot be used to prove that a language is Regular.

If A is a Regular language, then A has a pumping Length 'p' such that any string 'S' where |S|>=P may be divided into 3 parts S=xyz such that the following conditions must be true.

(1) $xy^iz \varepsilon A$ for any every i>=0

(2) |y| >0

(3) |xy|<=p

CHAPTER THREE

3. Context Free Language

Unit Contents:

3.1. Context Free Language

3.2. Context Free Grammar

3.3. Parsing and Derivation Tree

3.4. Pumping Lemma for Context Free Languages

3.5. Simplifications of Context Free Language

3.6. Normal Forms

3.1. Context Free Languages

A context-free language has important applications in the design of programming languages well as in the construction of efficient compilers.

Context Free Grammar: productions in a regular grammar are restricted in two ways:

the left side must be a single variable , the right side has a special form

By retaining the restriction on the left side, but permitting anything on the right, we get context free grammar.

Definition 3.1 A grammar G=(V,T,S,P) is said to be context free if all productions in p have the form: $A \rightarrow x$ where $A \varepsilon V$ and $x \varepsilon (V \cup T)^*$

A language L is said to be context free if and only if there is a context free grammar G such that L = L(G). Every regular grammar is context-free, so a regular language is also a context-free one.

Example: the grammar G = ({S},{a, b},S,P) with production

$S \rightarrow aSa$

$S \rightarrow bSb$

$S \rightarrow \lambda$ is context free. A typical derivation is this grammar is …

$S \rightarrow SaS \rightarrow aaSaa \rightarrow aabSbaa \rightarrow aabbaa$

L(G) =(wwR: w ε{a,b}*)

The grammar is not only context free but also linear

Example: consider the grammar with production

$S \rightarrow aSb|SS|\lambda$

This grammar is context free but not linear.

Leftmost and Rightmost Derivations

In context-free grammars that are not linear, a derivation may involve sentential forms with more than one variable.

Example: the grammar G = ({A, B, S},{a, b}, S, P) with production

$S \rightarrow AB$

$A \rightarrow aaA$

$A \rightarrow \lambda$

$B \rightarrow Bb$

$B \rightarrow \lambda$

$L(G) = \{a2nbm: n \geq 0, m \geq 0\}$

Consider the following derivations $S \rightarrow AB \rightarrow aaAB \rightarrow aaB \rightarrow aaBb \rightarrow aab$

$S \rightarrow AB \rightarrow ABb \rightarrow aaABb \rightarrow aaAb \rightarrow aab$

***Definition** 3.2*. the derivation is said to be leftmost if in each step the **leftmost** variable in the sentential form is replaced.

If in each step the right most variable is replaced, we call the derivation **rightmost**.

Example: consider the grammar with productions

$S \rightarrow aAB$

$A \rightarrow bBb$

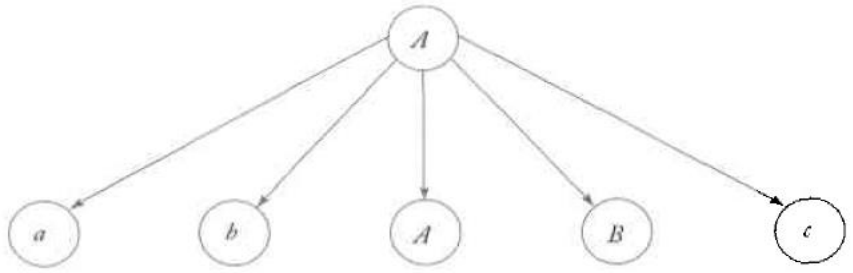
$B \rightarrow A|\lambda$

$S \rightarrow aAB \rightarrow abBbB \rightarrow abAbB \rightarrow abbBbbB \rightarrow abbbbB \rightarrow abbbb$ is leftmost derivation of the string abbbb. A rightmost derivation of the same string

$S \rightarrow aAB \rightarrow aA \rightarrow abBb \rightarrow abAb \rightarrow abbBbb \rightarrow abbbb$

1.2.**Derivation Tree**: show derivations independent of the order in which production are used. It is ordered tree in which nodes are labeled with the left sides of productions. The child of a node represents its corresponding right sides.

Example: $A \rightarrow abABc$ the derivation tree representing the production



Let G = (V, T, S, P) be a context-free grammar. An ordered tree is a derivation tree for G if and only if it has the following property.

1. The root is labeled S

2. Every leaf has label from T U {λ}

3. Every interior vertex (a vertex which is not a leaf) has a label from V.

4. If a vertex has label A ε V, and its children are labeled (from left to right) a1, a2, …, an, then P must contain a production

A → a1a2…an

5. A leaf labeled λ has no siblings, that is, a vertex with a child labeled λ can have no other children.

A tree that has properties 3, 4 and 5, but in which 1 doesn't necessarily hold and in which property 2 is replace by:

2. Every leaf has label from V U T U {λ}, is said to be a partial derivation tree

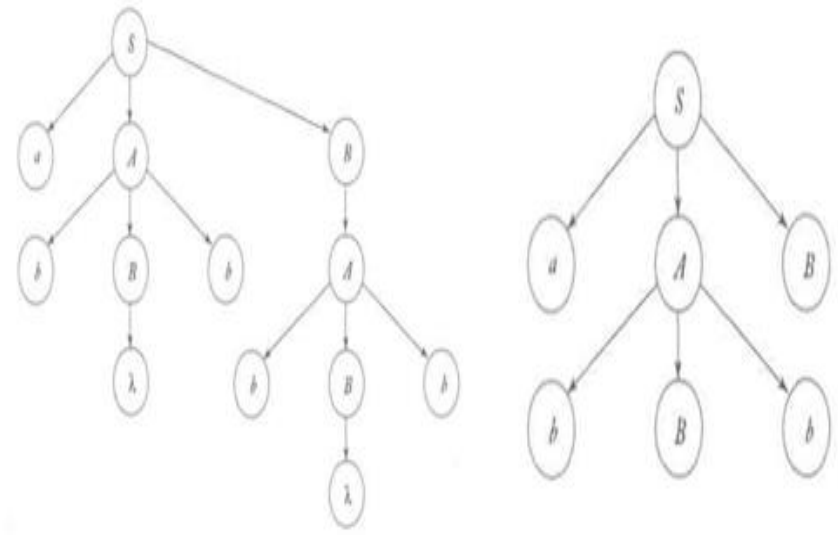
The string of symbols obtained by reading the leaves of the tree from left to right, omitting and λ's encountered, is said to be the yield of the tree.

Example: consider grammar G , with productions

S → aAB,

A → bBb

B → A|λ …… the string abBbB is derivation of the second and it is the sentential form of G. The yield of the second tree is the abbbb is the sentence of L (G).



1.3.Parsing and Ambiguity

The term parsing describes finding a sequence of productions by which a w ε L (G) is derived.

Given a string w in L (G), we can parse it in a rather obvious fashion; we systematically construct all possible (say, leftmost) derivations and see whether any of them matches w.

Example: consider the grammar S → SS|aSb| bSa|λ and the string w = aabb. Using exhaustive search parsing (Top-down parsing) we can solve as follow.

Round one

1.S → SS

2.S → aSb

3.S → bSa

4.S → λ

We can remove the last two

Round two: yields sentential forms

5.S → SS → SS

6.S → SS → aSbS

7.S → SS → bSaS

8.S → SS → S

From 2 we can get additional sentential forms

S → aSb → aSSb

S → aSb → aaSbb

S → aSb → abSab

S → aSb → ab

On the next round we get the target string

$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aabb$$

Therefore, aabb is the language generated by the given grammar.

Exhaustive parsing has serious flaws

- It is tedious
- it is not to be used where efficient parsing is required
- Never terminates for strings not in L(G)

In fact to tackle non termination of exhaustive search parsing we have to rule out the production of the form

- $A \rightarrow \lambda$
- $A \rightarrow B$

Theorem 3.2. suppose that G = (V, T, S, P) is a context free grammar which does not have any rule of the form:

- $A \rightarrow \lambda$
- $A \rightarrow B$ where A B ε V, the exhaustive parsing either produces parsing of w, or tell as no parsing is possible.

Theorem 3.3. for ever context free grammar there exist an algorithm that parses any w ε L(G) in a number of steps proportional to $|w|^3$

Definition 3.4. A context-free grammar G = (V, T, S, P) is said to be a simple grammar or s-grammar if all its production is of the form.

$A \rightarrow ax$ where A ε V, a ε T, x ε V* and any pair(A, a) occurs at most once in the production

Example: the grammar

$S \rightarrow aS|bSS|c$ is a simple grammar

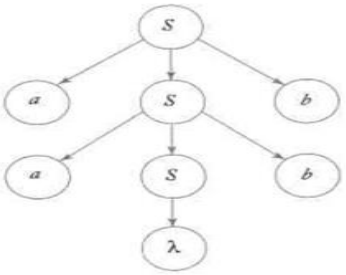
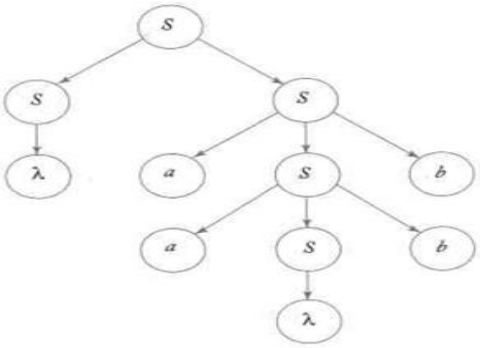
Definition: A context-free grammar G is said to be ambiguous if there exists some w ε L (G) that has at least two distinct derivation trees‘.

Ambiguity implies the existence of two or more leftmost or rightmost derivations.

Example: the grammar with production

$S \rightarrow aSb|SS|\lambda$, is ambiguous

The sentence aabb has two derivation trees

Consider the grammar G = (V, T, E, P) with V= {E, I} and T= {a, b, c, +, *, (,)} and productions.

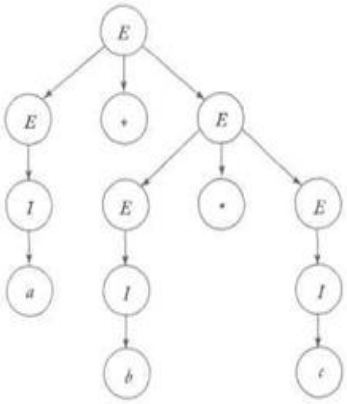
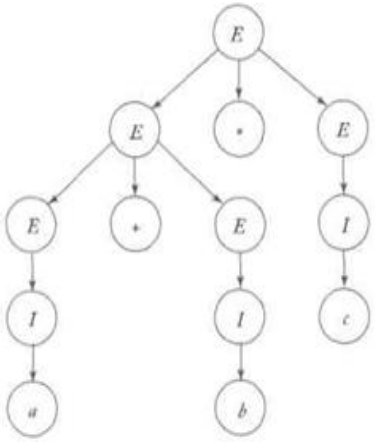
$E \rightarrow I,$

$E \rightarrow E + E,$

$E \rightarrow E * E,$

$E \rightarrow (E),$

$I \rightarrow a|b|c.$ This grammar is ambiguous. a + b*c has two d/t derivation tree

The solution it to rewrite the grammar in which only one parsing is possible. The ambiguity came from the grammar in the sentence could be removed by finding an equivalent unambiguous grammar.

Definition 3.6. If L is a context free language for which there exist a grammar, then L is said to be unambiguous. If every grammar that generates L is ambiguous, then the language is inherently ambiguous.

1.4. Context Free Grammar and Programming Languages

One important use of theory of formal language is in the construction of compiler and interpreter. As with other languages a programming language can be defined using a grammar. Backus-Naur form (BNF) can be used for specifying grammars. In BNF variables are enclosed in parenthesis

Example: the grammar with this production

$E \rightarrow T$, $T \rightarrow T * F$,

$T \rightarrow F$, $F \rightarrow (E)$

$F \rightarrow I$, $I \rightarrow a|b|c.$

$E \rightarrow E + T$,

This can be represented in BNF as

<exprssion> := <term>|<expression> + <term>,

<term> := <factor> | <term> *<factor>

1.5. **Simplification of context free grammars and normal forms**

1. Substitution Rule:

Theorem 3.1. Let G= (V, T, S, P) be a context-free grammar. Suppose p contains the production of the form :

$A \rightarrow x1Bx2$ assume that A and B are different variables and that

$B \rightarrow y1|y2|\ldots yn$ is the set of the productions in P which have B at the left said.

Let G: = (V, T, S, P:) be the grammar in which P: is constructed by deleting $A \rightarrow x1Bx2$ form p and adding to it $A \rightarrow x1y1x2| x1y2x2 | \ldots | x1ynx2$ then

L(G:) = L(G)

Example 3.1. Consider G= ({A, B}, {a, b, c}, A, P) with production

$A \rightarrow a|aaA|abBc$

$B \rightarrow abbA|b$

Using the suggested substitution we get the grammar G: with productions

$A \rightarrow a|aaA|ababbAc|abbc$

the new grammar G: is the equivalent to G

Take the sentences aaabbc possible generated in the two grammars.

We will see shortly how much unnecessary productions can be removed from a grammar.

2. Removing Useless Productions

Remove productions from a grammar that can never take part in any derivation.

Example: $S \rightarrow aSb|\lambda|A$,

$A \rightarrow aA$,

This production clearly not part of the production since it cannot transformed into a terminal string.

Definition 3.1. Let G= (V, T, S, P) be a context-free grammar. A variable A ε V is said to be useful if and only if there is at least one w ε L(G) such that exist

$S \rightarrow xAy \rightarrow w$ with x, y ε(V UT) *

A variable s useful if and only if it occurs in at least one derivation.

A variable that is not useful is called useless. A

Production is useless if it involves any useless variable.

A variable may be useless because there is no way of getting a terminal string from it.

Example: See the production

$S \rightarrow A$,

$A \rightarrow aA|\lambda$

$B \rightarrow bA$, the variable B is useless in the production $B \rightarrow bA$. Although B can derive a terminal string, there is no way to achieve the definition.

Example: Eliminate useless symbols and production from G=(V, T, S, P) where V={S, A, B, C} and T={a, b} with P consisting of

$S \rightarrow aS|A|C$,

$A \rightarrow a$,

$B \rightarrow aa$,

$C \rightarrow aCb$

First find the set of variables that lead to the set of terminal strings. S, A and B belongs to this set but not C. Hence, remove C since it is useless

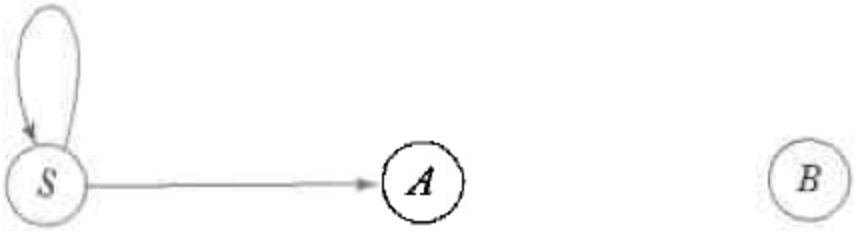
The resulted grammar G_1 has V={S, A, B} and T={a} and production

$S \rightarrow aS|A$,

$A \rightarrow a$,

$B \rightarrow aa$,

Then, eliminate the variable that cannot be reached from the start variable. Draw a dependency graph. A dependency graph has its vertices label with a variable with an edge between vertices C and D if and only if there is a production form $C \rightarrow xDy$. A dependency graph for the grammar is as shown



A variable is useful if there is a past from vertex S to itself. Therefore, B is useless. Therefore, the final answer set V:={S, A} T:={a} and production

$S \rightarrow aS|A,$

$A \rightarrow a,$

This result *Theorem 6.2.* for every context free grammar the is equivalent grammar that doesn't contain useless productions.

3. Removing λ production

A kind of production with empty string at the right side should be remove.

Definition 6.2: a production of CFG of the form

$A \rightarrow \lambda$ ------ is called λ production

Any variable A for which the derivation $A \rightarrow \lambda$ is called nullable. Both λ production and nullable should be removed.

Example: consider the grammar

- $S \rightarrow a\ S_1 b$
- $S_1 \rightarrow a\ S_1 b|\ \lambda$

λ free language can be generated as follow

- $S \rightarrow a\ S_1 b|ab,$
- $S_1 \rightarrow a\ S_1 b|\ ab.$

- This new grammar generates the same language as the original one.
- **Theorem 6.3.** Let G be a context free grammar with λ in L(G). Then, there exist a equivalent grammar $\widehat{G}$ having no λ-production.
  - First find the set of V_N of all nullabel variables of G, using the following step.
    1. For all production $A \rightarrow \lambda$, put A in to V_N
    2. Repeat the following step until no further variables are added to . For all productions:
       - $B \rightarrow A_1A_2....A_n$ where $A_1, A_2,....., A_n$ are in V_N put B in V_N

Example: Find CFG without λ-productions equivalent to the grammar defined by:

S → ABaC

A → BC

B→ b|λ,

C→ D|λ,

D → d.

Removing λ-productions we will get

S → ABaC|BaC|AaC|ABa|aC|Aa|Ba|a

A → B|C|BC

B→ b,

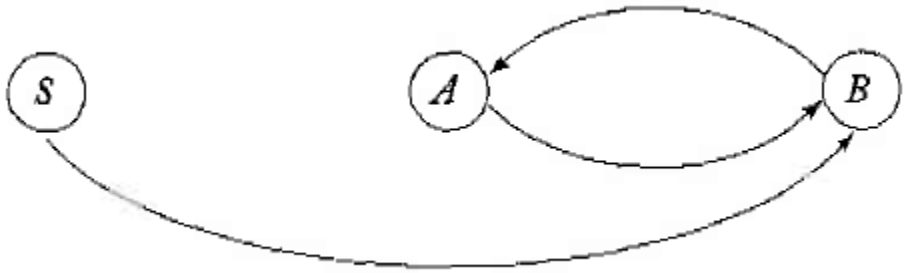
C→ D,

D → d.

4. **Removing unit production**

- Definition 6.3. any production of CFG of the form A → B where A and B ε V is called unit – production.
- To remove unit-production use substitution rule
- Theorem 6.4. Let G= (V, T, S, P) be any CFG without λ– productions. Then there exists a CFG $\hat{G} = (\hat{V}, \hat{T}, \hat{S}, \hat{P})$ that doesn't have unit- production and that is equivalent to G.
- Example: Remove all unit productions from
  - S → Aa | B
  - B → A | bb
  - A → a | bc | B

First construct a dependency graph.



S →A, S→B, B →A and A → B add original unit productions

S → Aa

A → a|bc

B → bb

The new rules are

S → a|bc|bb

A → bb

B → a|bc

The equivalent grammar

S → a|bc|bb|Aa

A → a|bb|bc

B → a|bb|bc|

3.6. The Two Important Normal Forms

Chomsky Normal form

- Symbols on the right of a production are strictly limited.
- String on the right of a production consist of no more than two symbols

Definition 6.4. A CFG is in Chomsky normal form if all productions are of the form:

A → BC or A → a where A, B and C are in V and a is in T

Example: the grammar

S → AS|a,

A → SA|b. is in Chomsky normal form but the grammar

S → AS|AAS

A → SA|aa is not.

Chomsky Normal

- Theorem 6.6. Any CFG G =(V, T, S, P) with λ not element of L(G) has an equivalent grammar $\hat{G} = (\hat{V}, \hat{T}, \hat{S}, \hat{P})$ in Chomsky form:
  - Proof:
    - Step 1: remove all terminal symbols length > 1
      - construct grammar $G_1 = (V_1, T, S, P_1)$ from G by considering all productions in P in the form.
      - $A \rightarrow x_1x_2....x_n$ where x_i is the symbol either in V or T.
      - If n=1 put this production in P_1 ,
      - If $n \geq 2$ introduce a new variable B_a for each a ε T.
      - For each P in the of form $A \rightarrow x_1x_2....x_n$ put into P_1 $A \rightarrow C_1C_2....C_n$ in this case $C_i=x_i$ if x_i is in V or C_i= Ba if x_i =a

For every B_a if P_1 also put the production $B_a \rightarrow a$

Step 2: Introduce additional symbols to reduce the length of the right side.

If n=2 put as it is

If n > 2 introduce new variable D_1, D_2, …. D_n and put into productions.

$A \rightarrow C_1D_1$

$D_1 \rightarrow C_2D_2$

. ……..

……

$D_{n-2} \rightarrow C_{n-1}C_n$

Example: Convert the grammar with productions

$S \rightarrow ABa,$

$A \rightarrow aab,$

$B \rightarrow Ac.$ To Chomsky normal form.

Step 1:

$S \rightarrow ABB_a,$ $B_a \rightarrow a,$

$A \rightarrow B_aB_aB_b,$ $B_b \rightarrow b,$

$B \rightarrow AB_c$ $B_c \rightarrow c.$

Step 2: introduce additional variables to get the first two into normal form.

$S \rightarrow AD_1$ $B \rightarrow AB_c$

$D_1 \rightarrow BB_a,$ $B_a \rightarrow a,$

$A \rightarrow B_aD_2$ $B_b \rightarrow b,$

$D_2 \rightarrow B_aB_b,$ $B_c \rightarrow c.$

Greibach Normal Forms

Definition: a context-free grammar G is said to be in GNF if all productions have the form

$A \rightarrow ax$, where $a \varepsilon T$ and $x \varepsilon V^*$.

Example: the grammar $S \rightarrow AB,$

$A \rightarrow aA/bB/b,$

$B \rightarrow b$

is not in GNF.However,using the substitution rule we get the equivalent grammar

S → aAB/bBB/bB,

A → aA/bB/b,

B → b, which is in a GNF.

CHAPTER FOUR

4. Pushdown Automata

Unit Contents

4.1. Introduction to Pushdown Automata

4.2. Pushdown Automata and Language

4.3. Context Sensitive Language

4.4. Deterministic and Nondeterministic Pushdown Automata

Introduction

A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically, a pushdown automaton is = **"Finite state machine" + "a stack".**

A pushdown automaton has three components:

- an input tape,
- a control unit, and
- a stack with infinite size.

The stack head scans the top symbol of the stack.

A stack does two operations:

- **Push** − a new symbol is added at the top.
- **Pop** − the top symbol is read and removed.

A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.

Pushdown Automata has extra memory called **stack** which gives more power than Finite automata. It is used to recognize context-free languages.

Deterministic and Non-Deterministic PDA: In deterministic PDA, there is only one move from every state on every input symbol but in Non-Deterministic PDA, there can be more than one move from one state for an input symbol.

Note:

- Power of NPDA is more than DPDA.

- It is not possible to convert every NPDA to corresponding DPDA.
- Language accepted by DPDA is a subset of language accepted by NPDA.
- The languages accepted by DPDA are called DCFL (Deterministic Context-Free Languages) which are a subset of NCFL (Non-Deterministic CFL) accepted by NPDA.

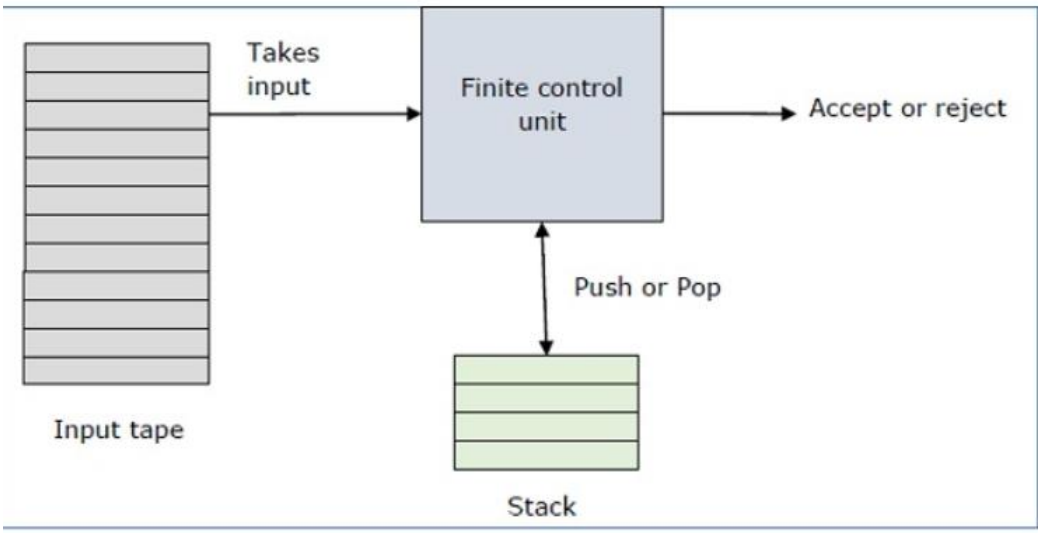
Finite automata cannot recognize all context-free languages, because finite automata have strictly finite memories allowing unbounded storage that is restricted to operating like a stack ==> pushdown automata

In this part we explore the connection between pda and cfl.

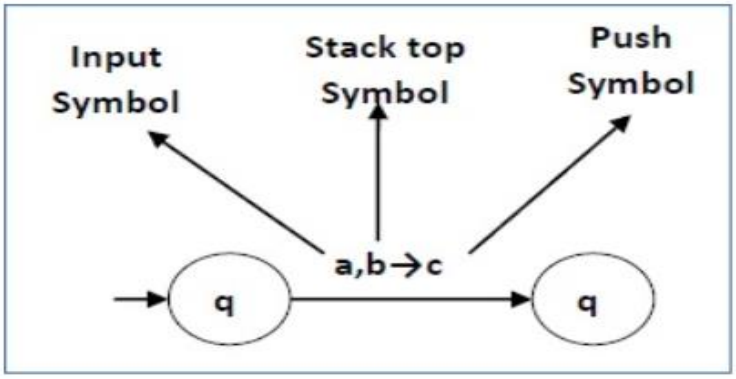
We have already discussed finite automata. But finite automata can be used to accept only regular languages. Pushdown Automata is a finite automata with extra memory called stack which helps Pushdown automata to recognize Context Free Languages.

A PDA can be formally described as a 7-tuple ($Q, \sum, S, \delta, q_0, I, F$):

- Q is the set of states
- $\sum$ is the set of input symbols
- Γ is the set of pushdown symbols (which can be pushed and popped from stack)
- q0 is the initial state
- Z is the initial pushdown symbol (which is initially present in stack)
- F is the set of final states
- δ is a transition function which maps $Q \times \{ \sum \cup \varepsilon \} \times \Gamma$ into $Q \times \Gamma^*$. In a given state, PDA will read input symbol and stack symbol (top of the stack) and move to a new state and change the symbol of stack.



The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as $a,b \rightarrow c$:



This means at state q_1, if we encounter an input string **'a'** and top symbol of the stack is **'b'**, then we pop **'b'**, push **'c'** on top of the stack and move to state q_2.

Terminologies Related to PDA

Instantaneous Description

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where

q is the state

w is unconsumed input

s is the stack contents

Turnstile Notation

The "turnstile" notation is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transition is denoted by the turnstile symbol "⊢".

Consider a PDA $(Q, \sum, S, \delta, q_0, I, F)$. A transition can be mathematically represented by the following turnstile notation −

$$(p, aw, T\beta) \vdash (q, w, \alpha b)$$

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

Note − If we want zero or more moves of a PDA, we have to use the symbol (⊢*) for it.

Pushdown Automata Acceptance

There are two different ways to define PDA acceptability.

Final State Acceptability

In final state acceptability, a PDA accepts a string when, after reading the entire string, the PDA is in a final state. From the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state.

For a PDA $(Q, \sum, S, \delta, q_0, I, F)$, the language accepted by the set of final states F is −

$L(PDA) = \{w \mid (q_0, w, I) \vdash^* (q, \varepsilon, x), q \in F\}$

for any input stack string **x**.

Empty Stack Acceptability

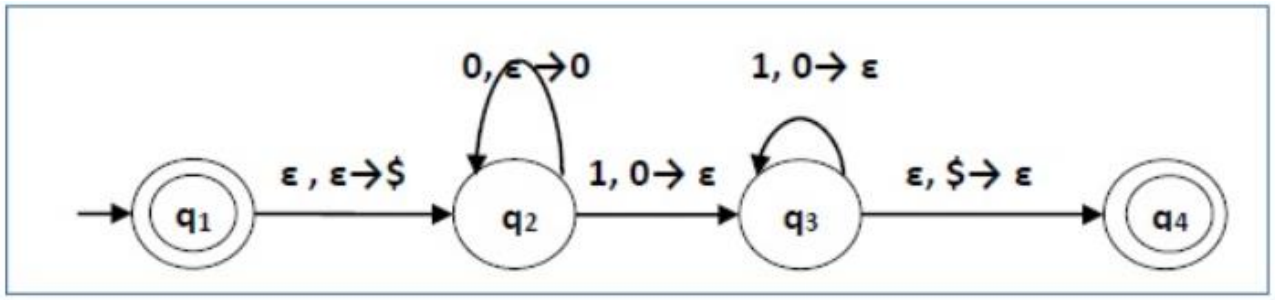
Here a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack.

For a PDA (Q, ∑, S, δ, q_0, I, F), the language accepted by the empty stack is −

L(PDA) = {w | (q_0, w, I) ⊢* (q, ε, ε), q ∈ Q}

Example: **Construct a PDA that accepts L = {$0^n 1^n$ | n ≥ 0}?**

Solutions



PDA for L= {$0^n 1^n$ | n≥0}

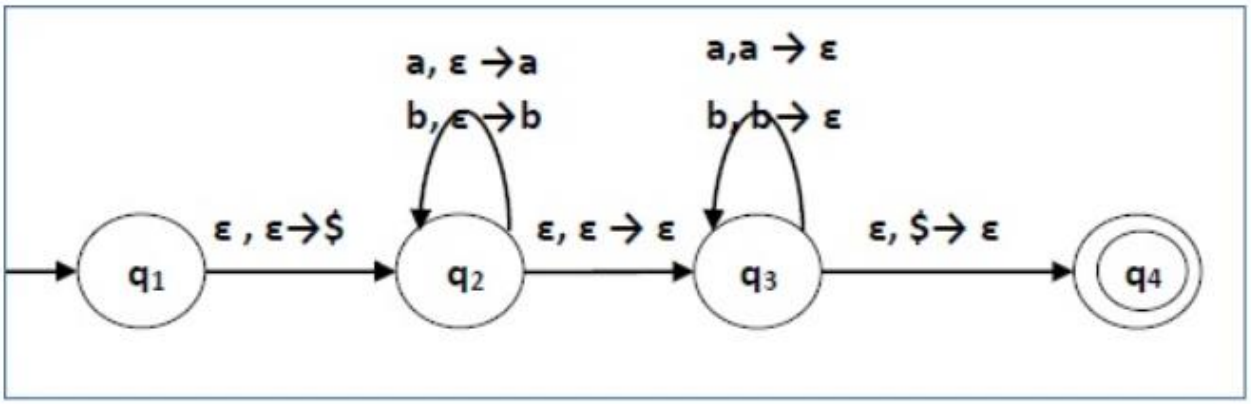
This language accepts L = {ε, 01, 0011, 000111, ... }

Here, in this example, the number of **'a'** and **'b'** have to be same.

Initially we put a special symbol **'$'** into the empty stack. Then at state **q_2**, if we encounter input 0 and top is Null, we push 0 into stack. This may iterate. And if we encounter input 1 and top is 0, we pop this 0. Then at state **q_3**, if we encounter input 1 and top is 0, we pop this 0. This may also iterate. And if we encounter input 1 and top is 0, we pop the top element. If the special symbol '$' is encountered at top of the stack, it is popped out and it finally goes to the accepting state q_4.

Example Construct a PDA that accepts L = { ww^R | w = (a+b)* }

Solution



PDA for L= {ww^R | w = (a+b)*}

Initially we put a special symbol '$' into the empty stack. At state q_2, the **w** is being read. In state q_3, each 0 or 1 is popped when it matches the input. If any other input is given, the PDA will go to a dead state. When we reach that special symbol '$', we go to the accepting state q_4.

Construct Pushdown Automata for given languages

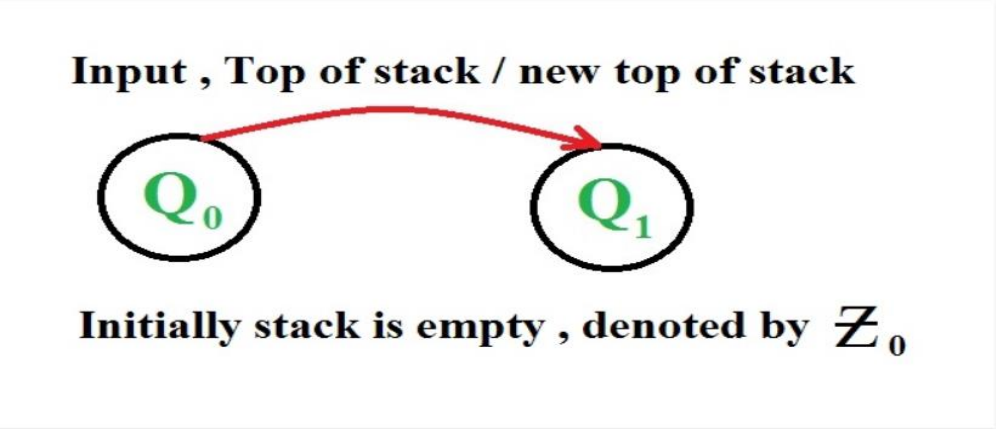
A push down automata is similar to deterministic finite automata except that it has a few more properties than a DFA. The data structure used for implementing a PDA is stack. A PDA has an output associated with every input. All the inputs are either pushed into a stack or just ignored. User can perform the basic push and pop operations on the stack which is use for PDA. One of the problems associated with DFAs was that could not make a count of number of characters which were given input to the machine. This problem is avoided by PDA as it uses a stack which provides us this facility also.

A Pushdown Automata (PDA) can be defined as –

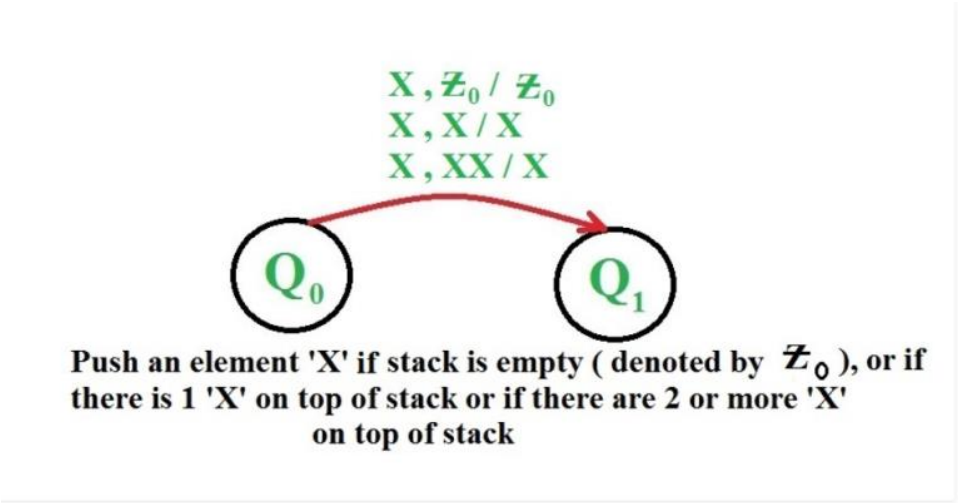
M = (Q, Σ, Γ, δ, q0, Z, F) where

- Q is a finite set of states
- Σ is a finite set which is called the input alphabet
- Γ is a finite set which is called the stack alphabet
- δ is a finite subset of Q X (Σ ∪ {ε} X Γ X Q X Γ^*) the transition relation.
- $q_0 \in Q$ is the start state
- $Z \in \Gamma$ is the initial stack symbol
- $F \subseteq Q$ is the set of accepting states

Representation of State Transition –

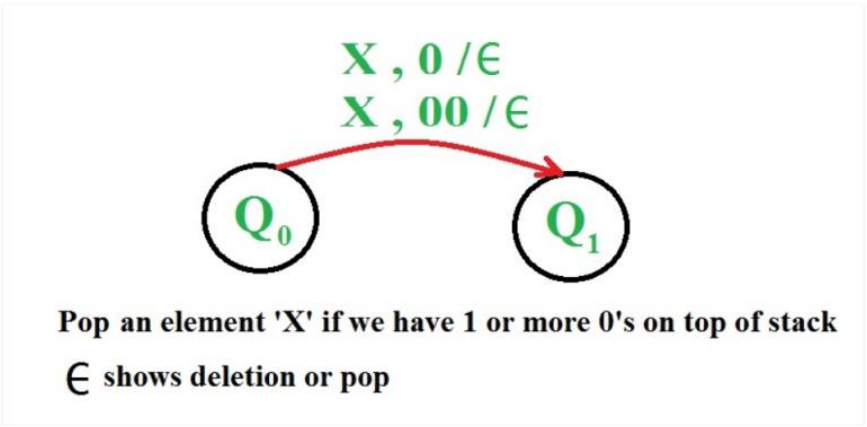


Representation of Push in a PDA:



Push an element 'X' if stack is empty (denoted by Z_0), or if there is 1 'X' on top of stack or if there are 2 or more 'X' on top of stack

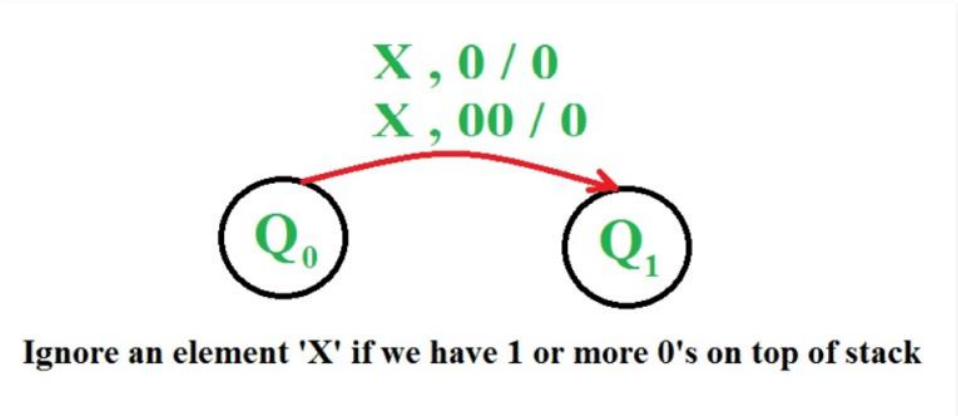
Representation of Pop in a PDA:



Pop an element 'X' if we have 1 or more 0's on top of stack

∈ shows deletion or pop

Representation of Ignore in a PDA:



Ignore an element 'X' if we have 1 or more 0's on top of stack

Q) Construct a PDA for language L = {$0^n1^m2^m3^n$ | n>=1, m>=1}

Approach used in this PDA –

First 0's are pushed into stack. Then 1's are pushed into stack.

Then for every 2 as input a 1 is popped out of stack. If some 2's are still left and top of stack is a 0 then string is not accepted by the PDA. Thereafter if 2's are finished and top of stack is a 0 then for

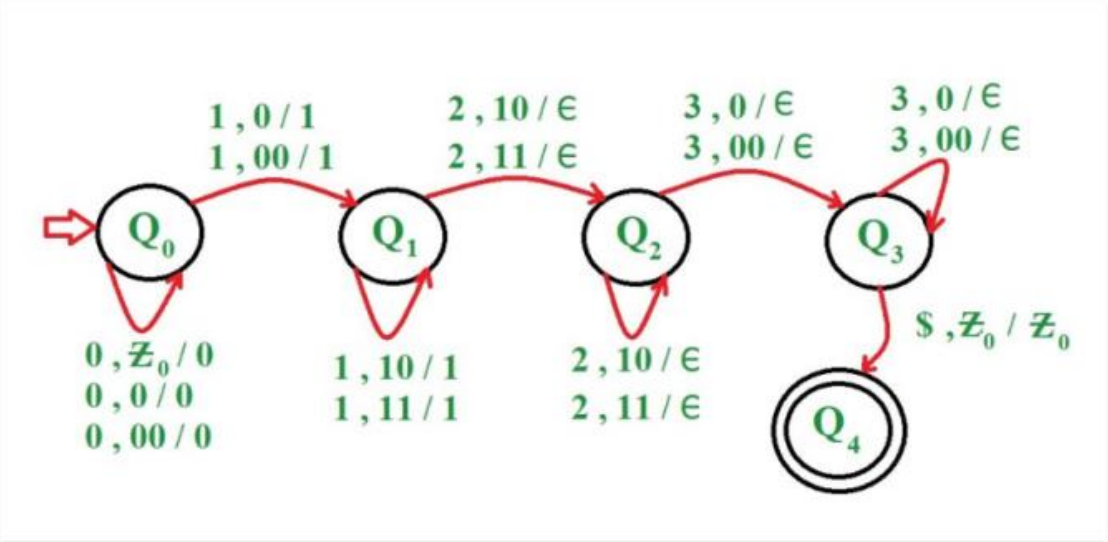
every 3 as input equal number of 0's are popped out of stack. If string is finished and stack is empty then string is accepted by the PDA otherwise not accepted.

Step-1: On receiving 0 push it onto stack. On receiving 1, push it onto stack and goto next state

Step-2: On receiving 1 push it onto stack. On receiving 2, pop 1 from stack and goto next state

Step-3: On receiving 2 pop 1 from stack. If all the 1's have been popped out of stack and now receive 3 then pop a 0 from stack and goto next state

Step-4: On receiving 3 pop 0 from stack. If input is finished and stack is empty then goto last state and string is accepted



Examples:

```
Input  : 0 0 1 1 1 2 2 2 3 3
Result : ACCEPTED
Input  : 0 0 0 1 1 2 2 2 3 3
Result : NOT ACCEPTED
```

Q) Construct a PDA for language L = {0^n1^m | n >= 1, m >= 1, m > n+2}

Approach used in this PDA –

First 0's are pushed into stack.When 0's are finished, two 1's are ignored. Thereafter for every 1 as input a 0 is popped out of stack. When stack is empty and still some 1's are left then all of them are ignored.

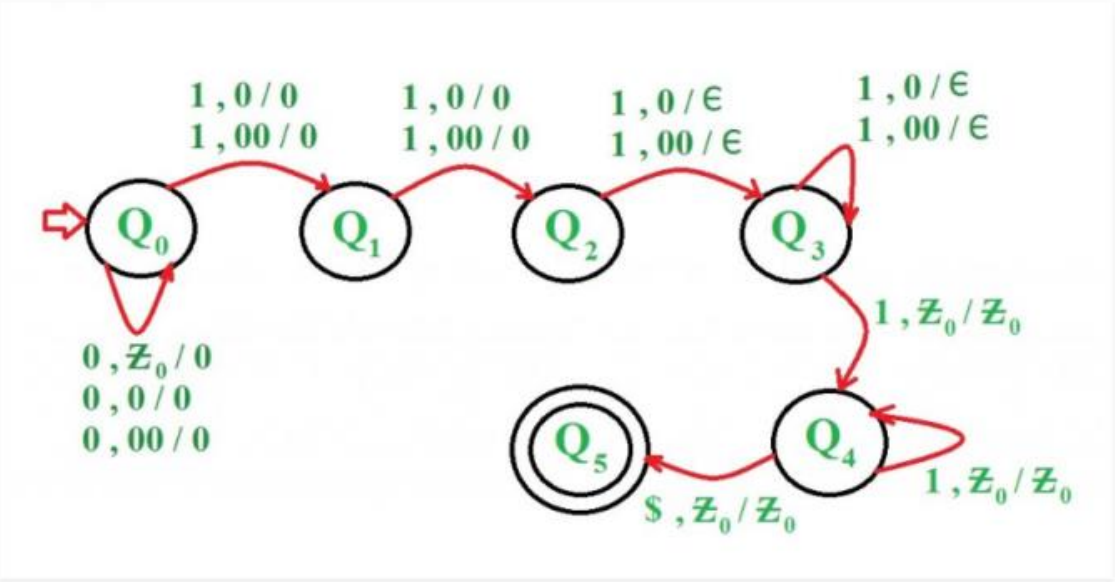
Step-1: On receiving 0 push it onto stack. On receiving 1, ignore it and goto next state

Step-2: On receiving 1, ignore it and goto next state

Step-3: On receiving 1, pop a 0 from top of stack and go to next state

Step-4: On receiving 1, pop a 0 from top of stack. If stack is empty, on receiving 1 ingore it and goto next state

Step-5: On receiving 1 ignore it. If input is finished then goto last state



Examples:

Input : 0 0 0 1 1 1 1 1 1
Result : ACCEPTED
Input : 0 0 0 0 1 1 1 1
Result : NOT ACCEPTED

Example : Define the pushdown automata for language $\{a^n b^n \mid n > 0\}$

Solution : M = where Q = { q0, q1 } and ∑ = { a, b } and Γ = { A, Z } and δ is given by :

δ(q0, a, Z) = { (q0, AZ) }

δ(q0, a, A) = { (q0, AA) }

δ(q0, b, A) = { (q1, ε) }

δ(q1, b, A) = { (q1, ε) }

δ(q1, ε, Z) = { (q1, ε) }

Let us see how this automata works for aaabbb.

Row	State	Input	δ (transition function used)	Stack(Leftmost symbol represents top of stack)	State after move
1	q0	aaabbb		Z	q0
2	q0	**a**aabbb	δ(q0,a,Z)={(q0,AZ)}	AZ	q0
3	q0	a**a**abbb	δ(q0,a,A)={(q0,AA)}	AAZ	q0
4	q0	aa**a**bbb	δ(q0,a,A)={(q0,AA)}	AAAZ	q0
5	q0	aaa**b**bb	δ(q0,b,A)={(q1,∈)}	AAZ	q1
6	q1	aaab**b**b	δ(q1,b,A)= {(q1,∈)}	AZ	q1
7	q1	aaabb**b**	δ(q1,b,A)= {(q1,∈)}	Z	q1
8	q1	∈	δ(q1, ∈,Z)= {(q1,∈)}	∈	q1

Explanation : Initially, the state of automata is q0 and symbol on stack is Z and the input is aaabbb as shown in row 1. On reading 'a' (shown in bold in row 2), the state will remain q0 and it will push symbol A on stack. On next 'a' (shown in row 3), it will push another symbol A on stack. After reading 3 a's, the stack will be AAAZ with A on the top. After reading 'b' (as shown in row 5), it will pop A and move to state q1 and stack will be AAZ. When all b's are read, the state will be q1 and stack will be Z. In row 8, on input symbol 'ε' and Z on stack, it will pop Z and stack will be empty. This type of acceptance is known as **acceptance by empty stack.**

Note :

The above pushdown automaton is deterministic in nature because there is only one move from a state on an input symbol and stack symbol.

The non-deterministic pushdown automata can have more than one move from a state on an input symbol and stack symbol.

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

Power of non-deterministic PDA is more as compared to deterministic PDA as some languages which are accepted by NPDA but not by deterministic PDA which will be discussed in next article.

The push down automata can either be implemented using empty stack or by final state and one can be converted to another.

Question : Which of the following pairs have DIFFERENT expressive power?

A. Deterministic finite automata(DFA) and Non-deterministic finite automata(NFA)

B. Deterministic push down automata(DPDA)and Non-deterministic push down automata(NPDA)

C. Deterministic single-tape Turing machine and Non-deterministic single-tape Turing machine
D. Single-tape Turing machine and multi-tape Turing machine

Solution : Every NFA can be converted into DFA. So, there expressive power is same. As discussed above, every NPDA can't be converted to DPDA. So, the power of NPDA and DPDA is not same. Hence option (B) is correct.

Pushdown Automata Acceptance by Final State

We have discussed Pushdown Automata (PDA) and its acceptance by empty stack article. Now, in this article, we will discuss how PDA can accept a CFL based on final state. Given a PDA P as:

$$P = (Q, \Sigma, \Gamma, \delta, q0, Z, F)$$

The language accepted by P is the set of all strings consuming which PDA can move from initial state to final state irrespective to any symbol left on the stack which can be depicted as:

$$L(P) = \{w \mid (q0, w, Z) \Rightarrow (qf, \varepsilon, s)\}$$

Here, from start state q0 and stack symbol Z, final state qf ε F is reached when input w is consumed. The stack can contain string s which is irrelevant as final state is reached and w will be accepted.

Example: Define the pushdown automata for language {a^nb^n | n > 0} using final state.
Solution: M = where Q = {q0, q1, q2, q3} and ∑ = {a, b} and Γ = { A, Z } and F={q3} and δ is given by:

$\delta(q0, a, Z) = \{ (q1, AZ) \}$

$\delta(q1, a, A) = \{ (q1, AA) \}$

$\delta(q1, b, A) = \{ (q2, \varepsilon) \}$

$\delta(q2, b, A) = \{ (q2, \varepsilon) \}$

$\delta(q2, \varepsilon, Z) = \{ (q3, Z) \}$

Let us see how this automata works for aaabbb:

Row	State	Input	δ (transition function) used	Stack (Leftmost symbol represents top of stack)	State after move
0	q0	aaabbb		Z	
1	q0	**a**aabbb	δ(q0,a,Z)={(q1,AZ)}	AZ	q1
2	q1	a**a**abbb	δ(q1,a,A)={(q1,AA)}	AAZ	q1
3	q1	aa**a**bbb	δ(q1,a,A)={(q1,AA)}	AAAZ	q1
4	q1	aaa**b**bb	δ(q1,b,A)={(q2, ε)}	AAZ	q2
5	q2	aaab**b**b	δ(q2,b,A)={(q2,ε)}	AZ	q2
6	q2	aaabb**b**	δ(q2,b,A)={(q2,ε)}	Z	q2
7	q2	ε	δ(q2,ε,Z)={(q3,ε)}	Z	q3

Explanation: Initially, the state of automata is q0 and symbol on stack is Z and the input is aaabbb as shown in row 0. On reading a (shown in bold in row 1), the state will be changed to q1 and it will push symbol A on stack. On next a (shown in row 2), it will push another symbol A on stack and remain in state q1. After reading 3 a's, the stack will be AAAZ with A on the top.

After reading b (as shown in row 4), it will pop A and move to state q2 and stack will be AAZ. When all b's are read, the state will be q2 and stack will be Z. In row 7, on input symbol ε and Z on stack, it will move to q3. As final state q3 has been reached after processing input, the string will be accepted.

This type of acceptance is known as *acceptance by final state*.

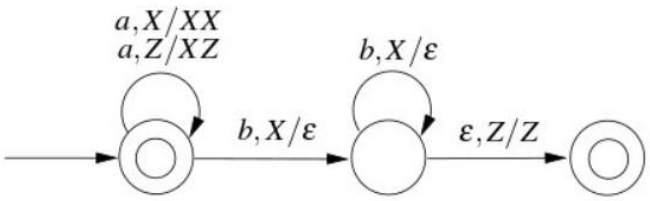
Next we will see how this automata works for aab:

Row	State	Input	δ (transition function) used	Stack (Leftmost symbol represents top of stack)	State after move
0	q0	aab		Z	
1	q0	**a**ab	δ(q0,a,Z)={(q1,AZ)}	AZ	q1
2	q1	a**a**b	δ(q1,a,A)={(q1,AA)}	AAZ	q1
3	q1	aa**b**	δ(q1,b,A)={(q2,ε)}	AZ	q2
4	q2	ε		AZ	

As we can see in row 4, the input has been processed and PDA is in state q2 which is non-final state, the string aab will not be accepted.

Let us discuss question based on this:

Que-1. Consider the transition diagram of a PDA given below with input alphabet ∑ = {a, b}and stack alphabet Γ = {X, Z}. Z is the initial stack symbol. Let L denote the language accepted by the PDA. (GATE-CS-2016)



Which one of the following is **TRUE**?

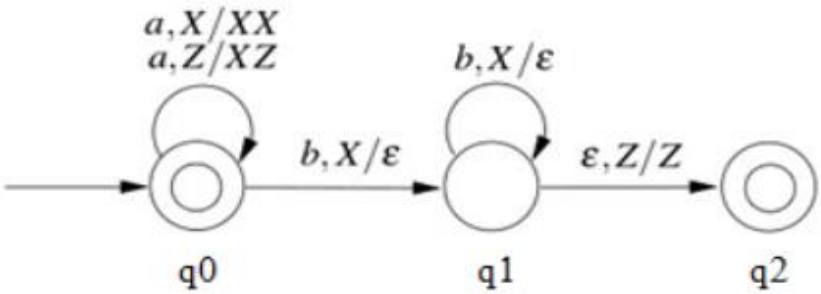
(A) $L = \{a^n b^n | n \geq 0\}$ and is not accepted by any finite automata

(B) $L = \{a^n | n \geq 0\} \cup \{a^n b^n | n \geq 0\}$ and is not accepted by any deterministic PDA

(C) L is not accepted by any Turing machine that halts on every input

(D) $L = \{a^n | n \geq 0\} \cup \{a^n b^n | n \geq 0\}$ and is deterministic context-free

Solution: We first label the state of given PDA as:



Next, the given PDA P can be written as:

Q = {q0, q1, q2} and ∑ = {a, b}

And Γ = {A, Z} and F={q0,q2} and δ is given by :

δ(q0, a, Z) = {(q0, XZ)}

δ(q0, a, X) = {(q0, XX)}

δ(q0, b, X) = {(q1, ε)}

δ(q1, b, X) = {(q1, ε)}

δ(q1, ε, Z) = {(q2, Z)}

As we can see, q0 is initial as well as final state, ε will be accepted. For every a, X is pushed onto stack and PDA remains in final state. Therefore, any number of a's can be accepted by PDA.

If input contains b, X is popped from stack for every b. Then PDA is moved to final state if stack becomes empty after processing input (δ(q1, ε, Z) = {(q2, Z)}). Therefore, number of b must be equal to number of b if they exist.

As there is only one move for a given state and input, the PDA is deterministic. So, correct option is (D).

Construct Pushdown Automata for all length palindrome

A Pushdown Automaton (PDA) is like an epsilon Non deterministic Finite Automata (NFA) with infinite stack. PDA is a way to implement context free languages. Hence, it is important to learn, how to draw PDA.

Here, take the example of odd length palindrome:

Que-1: Construct a PDA for language L = {wcw' | w={0, 1}*} where w' is the reverse of w.

Approach used in this PDA –

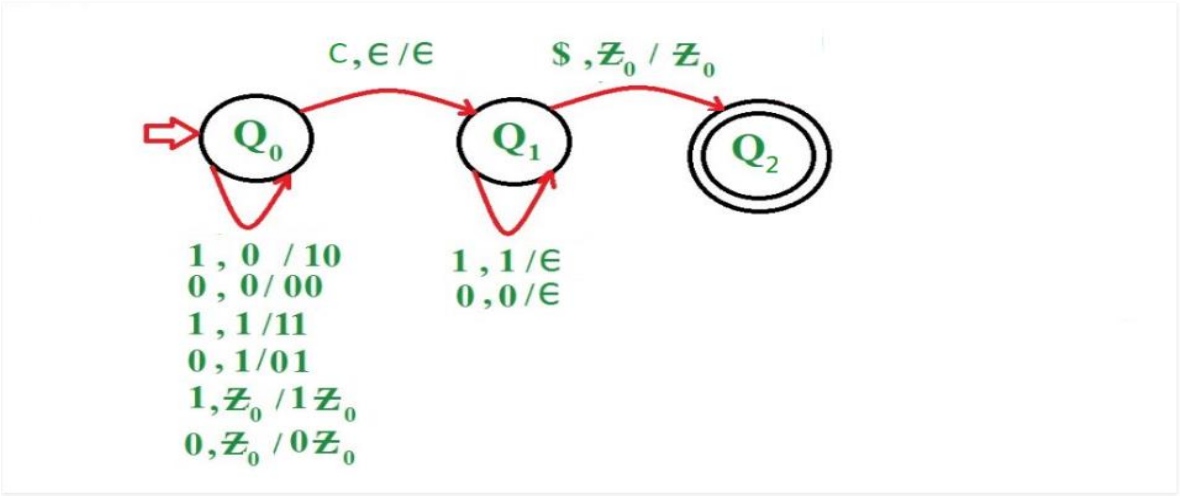
Keep on pushing 0's and 1's no matter whatever is on the top of stack until reach the middle element. When middle element 'c' is scanned then process it without making any changes in stack. Now if scanned symbol is '1' and top of stack also contain '1' then pop the element from top of stack or if scanned symbol is '0' and top of stack also contain '0' then pop the element from top of stack. If string becomes empty or scanned symbol is '$' and stack becomes empty, then reach to final state else move to dead state.

Step 1: On receiving 0 or 1, keep on pushing it on top of stack without going to next state.

Step 2: On receiving an element 'c', move to next state without making any change in stack.

Step 3: On receiving an element, check if symbol scanned is '1' and top of stack also contain '1' or if symbol scanned is '0' and top of stack also contain '0' then pop the element from top of stack else move to dead state. Keep on repeating step 3 until string becomes empty.

Step 4: Check if symbol scanned is '$' and stack does not contain any element then move to final state else move to dead state.



Examples:

Input : 1 0 1 0 1 0 1 0 1
Output :ACCEPTED
Input : 1 0 1 0 1 1 1 1 0
Output :NOT ACCEPTED

Now, take the example of even length palindrome:

Que-2: Construct a PDA for language L = {ww' | w={0, 1}*} where w' is the reverse of w.

Approach used in this PDA –

For construction of even length palindrome, user has to use Non Deterministic Pushdown Automata (NPDA). A NPDA is basically an NFA with a stack added to it.

The NPDA for this language is identical to the previous one except for epsilon transition. However, there is a significant difference, that this PDA must guess when to stop pushing symbols, jump to the final state and start matching off of the stack. Therefore this machine is decidedly non-deterministic.

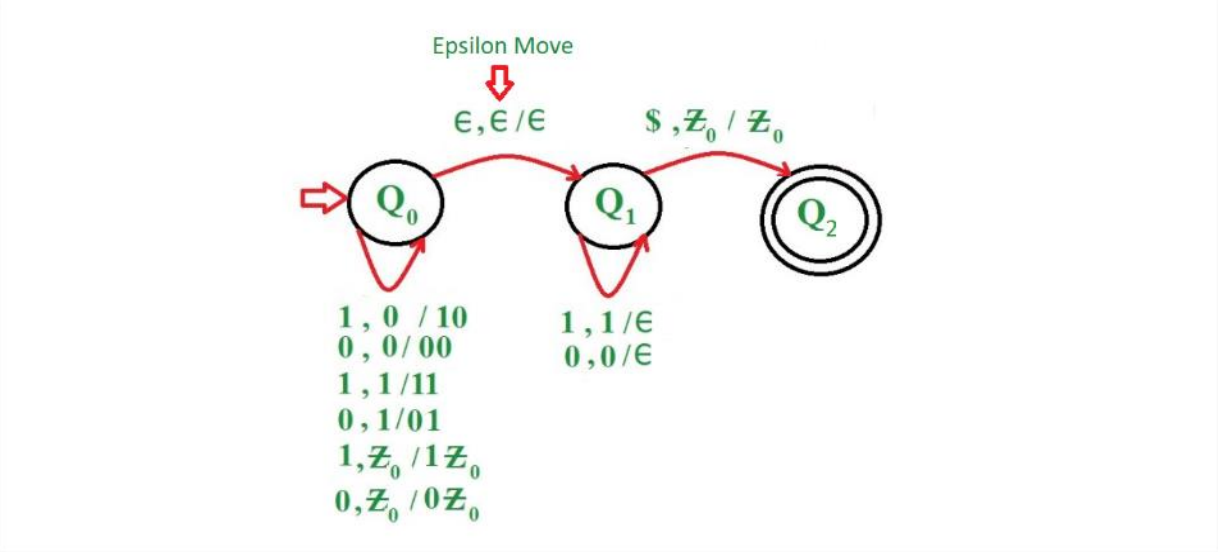
Keep on pushing 0's and 1's no matter whatever is on the top of stack and at the same time keep a check on the input string, whether reach to the second half of input string or not. If reach to last element of first half of the input string then after processing the last element of first half of input string make an epsilon move and move to next state. Now if scanned symbol is '1' and top of stack also contain '1' then pop the element from top of stack or if scanned symbol is '0' and top of stack also contain '0' then pop the element from top of stack. If string becomes empty or scanned symbol is '$' and stack becomes empty, then reach to final state else move to dead state.

Step 1: On receiving 0 or 1, keep on pushing it on top of stack and at a same time keep on checking whether reach to second half of input string or not.

Step 2: If reach to last element of first half of input string, then push that element on top of stack and then make an epsilon move to next state.

Step 3: On receiving an element, check if symbol scanned is '1' and top of stack also contain '1' or if symbol scanned is '0' and top of stack also contain '0' then pop the element from top of stack else move to dead state. Keep on repeating step 3 until string becomes empty.

Step 4: Check if symbol scanned is '$' and stack does not contain any element then move to final state else move to dead state.



Examples:

Input : 1 0 0 1 1 1 1 0 0 1
Output :ACCEPTED
Input : 1 0 0 1 1 1
Output :NOT ACCEPTED

Now, take the example of all length palindrome, i.e. a PDA which can accept both odd length palindrome and even length palindrome:

Que-3: Construct a PDA for language L = {ww' | wcw', w={0, 1}*} where w' is the reverse of w.

Approach used in this PDA –

For construction of all length palindrome, user has to use NPDA.

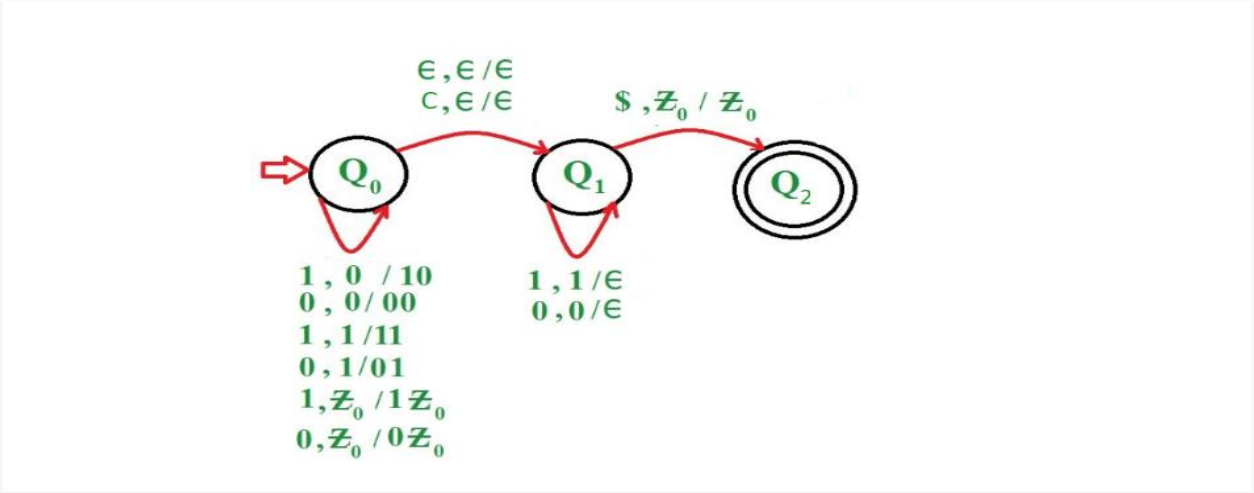
The approach is similar to above example, except now along with epsilon move now user has to show one more transition move of symbol 'c' i.e. if string is of odd length and if reach to middle element 'c' then just process it and move to next state without making any change in stack.

Step 1: On receiving 0 or 1, keep on pushing it on top of stack and at a same time keep on checking, if input string is of even length then whether reach to second half of input string or not, however if the input string is of odd length then keep on checking whether reach to middle element or not.

Step 2: If input string is of even length and reach to last element of first half of input string, then push that element on top of stack and then make an epsilon move to next state or if the input string is of odd length then on receiving an element 'c', move to next state without making any change in stack.

Step 3: On receiving an element, check if symbol scanned is '1' and top of stack also contain '1' or if symbol scanned is '0' and top of stack also contain '0' then pop the element from top of stack else move to dead state. Keep on repeating step 3 until string becomes empty.

Step 4: Check if symbol scanned is '$' and stack does not contain any element then move to final state else move to dead state.



Examples:

Input : 1 1 0 0 1 1 1 1 0 0 1 1
Output :ACCEPTED
Input : 1 0 1 0 1 0 1
Output :ACCEPTED

NPDA for accepting the language L = $\{a^m b^n c^{(m+n)} \mid m,n \geq 1\}$

Problem – Design a non deterministic PDA for accepting the language L = $\{a^m b^n c^{(m+n)} \mid m,n \geq 1\}$, i.e.,

L = {abcc, aabccc, abbbcccc, aaabbccccc, ...}

In each of the string, the total sum of the number of ‘a’ and ‘b’ is equal to the number of c’s. And all c’s are come after ‘a’ and ‘b’.

Explanation –

Here, we need to maintain the order of a’s, b’s and c’s.That is, all the a’s are are coming first and then all the b’s are coming after that all the c’s. Thus, we need a stack along with the state diagram. The count of a’s, b’s and c’s is maintained by the stack.
We will take 2 stack alphabets:

= { a, z }

Where, Γ = set of all the stack alphabet

z = stack start symbol

Approach used in the construction of PDA –

As we want to design a NPDA, thus every time ‘a’ comes before ‘b’ and ‘b’ comes before ‘c’. When ‘a’ comes then push it in stack and if again ‘a’ comes then also push it. After that, when ‘b’ comes then push ‘a’ into the stack and if again ‘b’ comes then also push it. And then when ‘c’ comes, pop one ‘a’ from the stack each time.

So, at the end if the stack becomes empty then we can say that the string is accepted by the PDA.

Stack transition functions –

(q0, a, z) ⊢(q0, az)

(q0, a, a) ⊢ (q0, aa)

(q0, b, a) ⊢ (q1, aa)

(q1, b, a) ⊢ (q1, aa)

(q1, c, a) ⊢ (q2,)

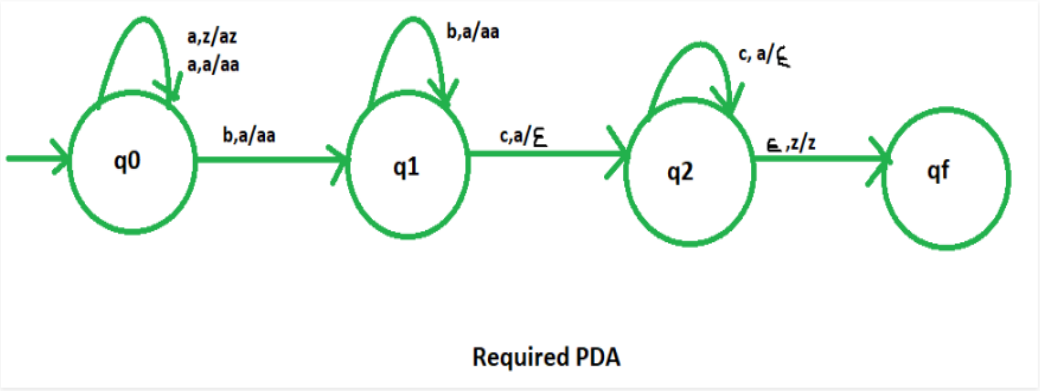
(q2, c, a) ⊢ (q2,)

(q2, , z) ⊢ (qf, z)

Where, q0 = Initial state

qf = Final state

= indicates pop operation



Required PDA

Construct Pushdown automata for L = $\{0^n1^m2^{(n+m)} \mid m,n \geq 0\}$

PDA plays a very important role in task of compiler designing. That is why there is a need to have a good practice on PDA. Our objective is to construct a PDA which accepts a string of the form {(0^n)(1^m)(2^(n+m))}

Example-

Input: 00001112222222
Output: Accepted

Input: 00011112222
Output: Not Accepted

Approach used in this PDA –

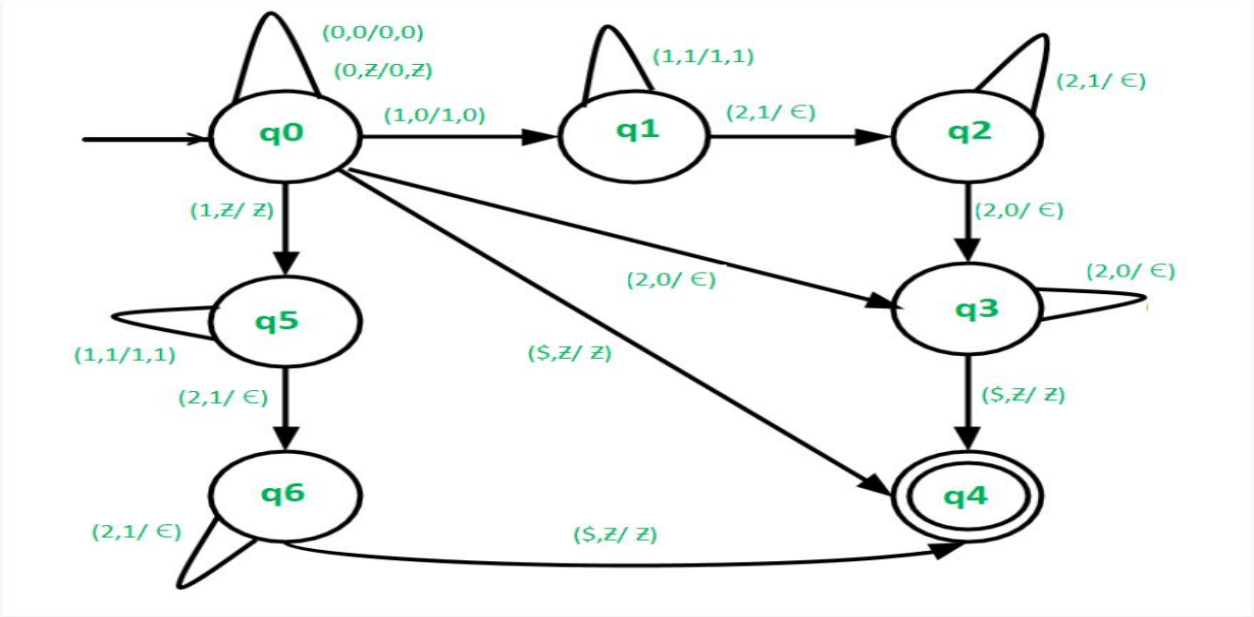
There can be four cases while processing the given input string.

Case 1- m=0: In this cases the input string will be of the form $\{0^n2^n\}$. In this condition, keep on pushing 0's in the stack until we encounter with 2. On receiving 2 check if top of stack is 0, then pop it from the stack. Keep on popping 0's until all the 2's of the string are processed. If we reach to the end of input string and stack becomes empty, then reached to the final state i.e. Accepts the input string else move to dead state.

Case 2- n=0: In this cases the input string will be of the form $\{1^m2^m\}$. In this condition, keep on pushing 1's in the stack until we encounter with 2. On receiving 2 check if top of stack is 1, then pop it from the stack. Keep on popping 1's until all the 2's of the string are processed. If we reach to the end of input string and stack becomes empty, then reached to the final state i.e. Accepts the input string else move to dead state.

Case 3- m, n>0: In this cases the input string will be of the form $\{0^n1^m2^{(n+m)}\}$. In this condition, keep on pushing 0's and 1's in the stack until we encounter with 2. On receiving 2 check if top of stack is 1 or 0, then pop it (1 or 0) from the stack. Keep on poping 1's or 0's until all the 2's of the input string are processed. If we reach to the end of input string and stack becomes empty, then reach to final state i.e. accept the input string else move to dead state.

Case 4- m=0, n=0: In this case the input string will be empty. Therefore directly jump to final state.



Construct Pushdown automata for L = $\{a^{(2*m)}c^{(4*n)}d^nb^m \mid m,n \geq 0\}$

PDA plays a very important role in task of compiler designing. That is why there is a need to have a good practice on PDA. Our objective is to construct a PDA for L = $\{a^{(2*m)}c^{(4*n)}d^nb^m \mid m,n \geq 0\}$

Examples –

Input: aaccccdb

Output: Accepted

Input: aaaacccccccddbb

Output: Accepted

Input: acccddb

Output: Not Accepted

Approach used in this PDA –

There can be four cases while processing the given input string.

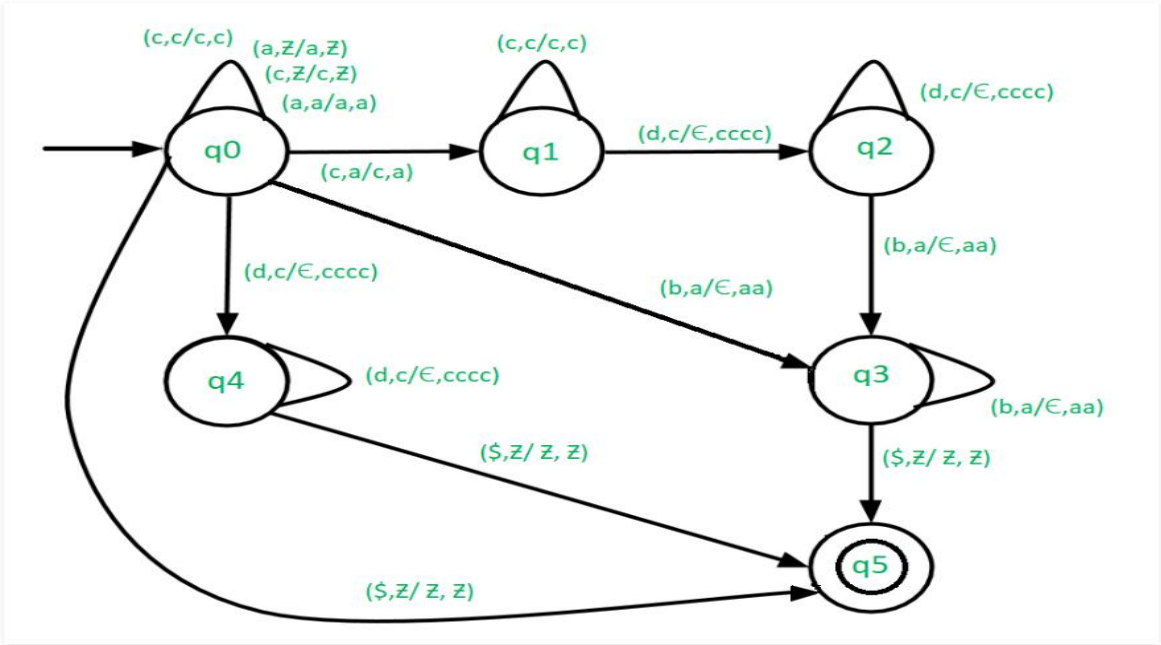
Case-1: m=0 – In this cases the input string will be of the form $\{c^{(4*n)}d^n\}$. In this condition, keep on pushing c's in the stack until we encounter with 'd'. On receiving 'd' check if top of stack is 'c',

then pop ‘cccc’ from the stack. Keep on popping cccc’s until all the d’s of the string are processed. If we reach to the end of input string and stack becomes empty, then reached to the final state, i.e., accepts the input string else move to dead state.

Case-2: n=0 – In this cases the input string will be of the form $\{a^{(2*m)}b^m\}$. In this condition, keep on pushing a’s in the stack until we encounter with ‘b’. On receiving b check if top of stack is ‘a’, then pop ‘aa’ from the stack. Keep on popping aa’s until all the b’s of the string are processed. If we reach to the end of input string and stack becomes empty, then reached to the final state i.e., accepts the input string else move to dead state.

Case-3: m, n>0 – In this cases the input string will be of the form $\{(a^{(2*m)}c^{(4*n)}d^nb^m\}$. In this condition, keep on pushing a’s and c’s in the stack until we encounter with ‘d’. On receiving d check if top of stack is ‘c’, then pop ‘cccc’ from the stack. Keep on poping cccc’s until all the d’s of the input string are processed. Then On receiving b check if top of stack is ‘a’, then pop ‘aa’ from the stack. Keep on poping aa’s until all the b’s of the input string are processed. If we reach to the end of input string and stack becomes empty, then reach to final state i.e., accept the input string else move to dead state.

Case-4: m, n=0 – In this case the input string will be empty. Therefore directly jump to final state.



Note –

1. We use (b, a/€, aa) to pop 2 a's, i.e., aa when encounter with b.
2. We use (d, c/€, cccc) to pop 4 c's, i.e., cccc when encounter with d.

NPDA for L = $\{0^i1^j2^k \mid i==j \text{ or } j==k ; i , j , k >= 1\}$

The language L = $\{0^i1^j2^k \mid i==j \text{ or } j==k ; i , j , k >= 1\}$ tells that every string of '0', '1' and '2' have certain number of 0's, then certain number of 1's and then certain number of 2's. The condition is that count of each of these 3 symbols should be atleast 1. Two important conditions for this language are that either count of 0 should be equal to count of 1 OR count of 1 should be equal to count of 2. Assume that string is ending with '$'.

Examples:

Input: 0 0 0 1 1 1 2 2 2 2 2
Here 0's = 3, 1's = 3 so i = j, 2's = 5
Output: Accepted
Input: 0 0 1 1 1 2 2 2
Here 0's = 2, 1's = 3, 2's = 3 so j = k
Output: Accepted
Input : 0 0 1 1 1 2 2 2 2
Here 0's = 2, 1's = 3, 2's = 4
Output: Not accepted

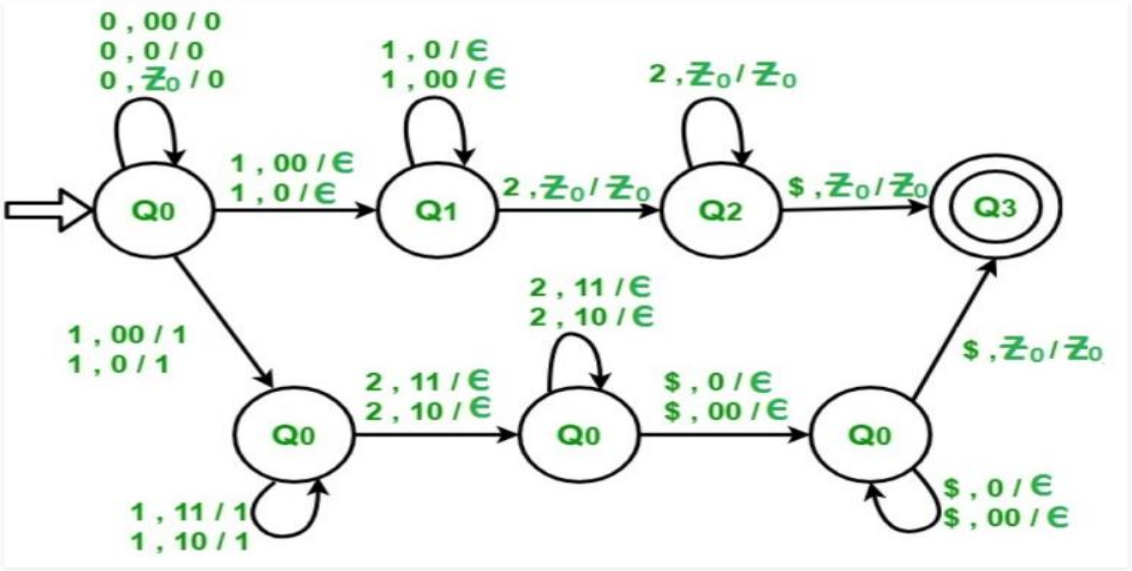
There are 2 approaches for the solution. First is for i==j and second is for j==k. These are:

Steps for i == j :

1. Input all 0's in the stack
2. When we get 1 as input pop a 0 from stack and goto next state.
3. If input is 1 then pop 0 from stack.
4. If stack becomes empty (i.e., every 0 corresponding to a 1 has been popped so i = j) and input is 2 then ignore it and goto next state.
5. If input is 2 then ignore it . If input is finished and $ is received then goto final state.

Steps for j == k :

1. Input all 0's in the stack
2. When we get 1 as input push it onto stack and goto next state.
3. If input is 1 then push it onto stack.
4. If input is 2 pop a 1 from stack and goto next state.
5. If input is 2 then pop 1 from stack. If input is finished and $ is received then pop a 0 from stack.
6. Pop all remaining 0's from the stack. If stack becomes empty then goto final state .



4.1. Nondeterministic pushdown Automaton

Definition: a non deterministic pushdown accepter (**npda**) is defined by a septuplet

M= {Q, Σ, Γ, δ, z, q_0, F),

Where

Q is a finite set of internal states of the control unit,

Σ is the input alphabets,

Γ is a finite set of symbols called the **stack alphabets**,

δ: QX (ΣU {λ}) X Γ → finite subsets of QX Γ* is the transition function,

q_0εQ is the initial state of the control unit,

Z ε Γ is the stack start symbols,

F c Q is the set of final states.

Example: the set of transition rules of an npda contains

δ (q1, a,b) = {(q2, cd), (q3, λ)}.

If at any time the control unit is in state q1, the input symbol read is a, and the symbol on the top of the stack is b, then one of the two things happen:

The control unit goes into state q2 and the string cd replaces b on the top of the stack.

The control unit goes into state q3 and b removed from the top of the stack.

Consider an npda with

Q = {q0, q1, q2, q3}

Σ = {a,b}

Γ ={0,1}

z= 0

F ={q3} and

δ(q0,a,0) = {(q1, 10), (q3, λ)}

δ(q0, λ,0) = {(q3, λ)}

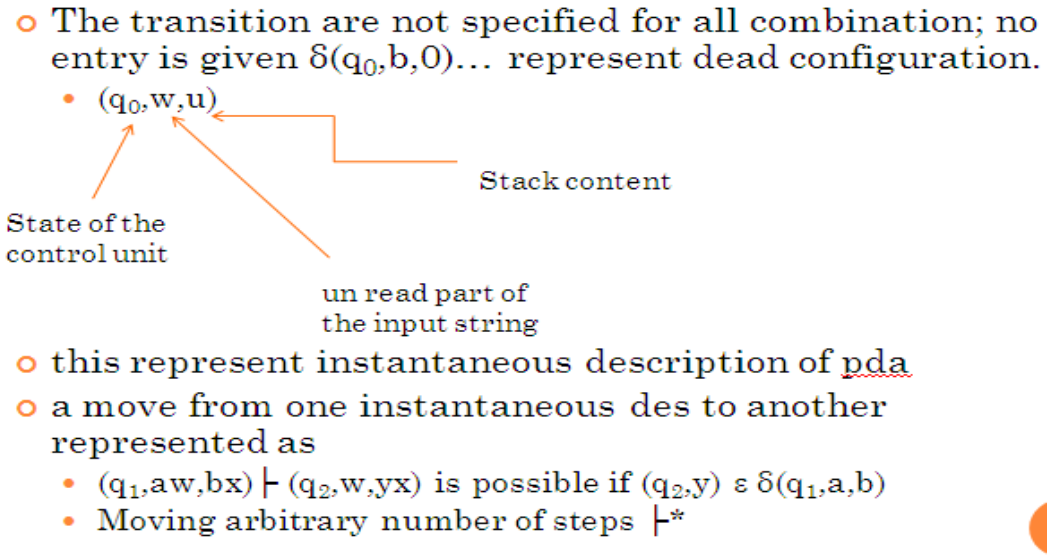
δ(q1,a,1) = {(q1, 11)}

δ(q1,b,1) = {(q2, λ)}

δ(q2,b,1) = {(q2, λ)}

δ(q2, λ,0) = {(q3, λ)}

What we can say about the action of this automaton?



A move from one instantaneous description to another will be denoted by the symbol ⊢

A move involving an arbitrary number of steps will be denoted by $\vdash^*$

4.2. The Language Accepted By a Pushdown Automaton

Definition: Let M= (Q, Σ, Γ, δ, z, q_0, F) be a nondeterministic pushdown automata.

4.4. Grammars for Deterministic Context-Free Language

Linear Bound Automata: Linear Bound Automata has a finite amount of memory called tape which can be used to recognize Context-Sensitive Languages.

 LBA is more powerful than Push down automata.

FA < PDA < LBA < TM

CHAPTER FIVE
5. Turing Machines

Unit contents

- ❖ Turing Machines
- ❖ Post Machines
- ❖ Variation on TM, TM encoding, Universal TM, Defining Computers by TM
- ❖ Chomsky's hierarchy of grammars Turing machines theory
- ❖ Decidability

1. Introduction to Turing Machine

Turing machine has infinite size tape and it is used to accept Recursive Enumerable Languages.

- Turing Machine can move in both directions. Also, it doesn't accept ε.
- If the string inserted in not in language, the machine will halt in the non-final state.

Deterministic and Non-Deterministic Turing Machines: In deterministic Turing machine, there is only one move from every state on every input symbol but in Non-Deterministic Turing machine, there can be more than one move from one state for an input symbol.

Note:

Language accepted by NTM, multi-tape TM and DTM are same.

Power of NTM, Multi-Tape TM and DTM is same.

Every NTM can be converted to corresponding DTM

A ***Turing machine*** (***tm***) is a finite automaton with a twist:

The tape head can move in either direction (left OR right)

A square on the input tape can be overwritten with another symbol

By the way, there is no stack!

A ***Turing machine*** (***tm***)

$$\boldsymbol{M} = (\boldsymbol{Q}, \Sigma, \Gamma, \delta, \square, \boldsymbol{q_0}, \boldsymbol{F}), \text{ where}$$

$\boldsymbol{Q}$ is a finite set of states,

is the input alphabet,

is a finite set of symbols (the ***tape alphabet***)

is the transition function

$: \boldsymbol{Q} \times \boldsymbol{\Gamma} \rightarrow \boldsymbol{Q} \times \boldsymbol{\Gamma} \times \{\boldsymbol{L}, \boldsymbol{R}\}$

$\boldsymbol{\Gamma}$ is a special symbol called the ***blank***

$= \Gamma - \{\square\}$

q_0 is the initial state

F is the set of final (accepting) states

A ***Turing machine*** (***tm***)

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

A tm starts in the initial state, generally looking at the leftmost non-blank symbol on the tape.

Accepter: The machine either accepts or rejects an input tape by whether it winds up in an accepting state or not.

Transducer: The machine computes, by modifying the input tape to produce an "output."

Notation: instantaneous description

Shows the contents of the tape and the position of the tape head at any given moment

$a_1a_2 \ldots a_{k-1}q_1a_ka_{k+1}\ldots a_n \text{¢} \; a_1a_2 \ldots a_{k-1}bq_2a_{k+1}\ldots a_n$

assuming that there is a transition

$\delta(q_1, a_k) = (q_2, b, R)$

Note: unlike our other automata, a Turing machine CAN go into an infinite loop!

Keep moving back and forth over a symbol without changing it!!!

x_1qx_2  ∞

What can a Turing machine do? Clearly, it can accept any regular language. Just limit the "instructions" to read from left to right and never change the contents of the input tape.

Can it accept the language $\{a^nb^n\}$? How?

- How about $\{wcw^R\}$?
- How about $\{ww^R\}$?
- How about $\{a^nb^nc^n\}$?
- How about $\{ww\}$?

In order to build a Turing machine, we need to develop an overall strategy. This can usually be stated in simple English by describing how the tm will move around and mark the input tape.

Let's look at some of these examples.

Can it accept the language $\{a^nb^n\}$? How?

Lay out a strategy

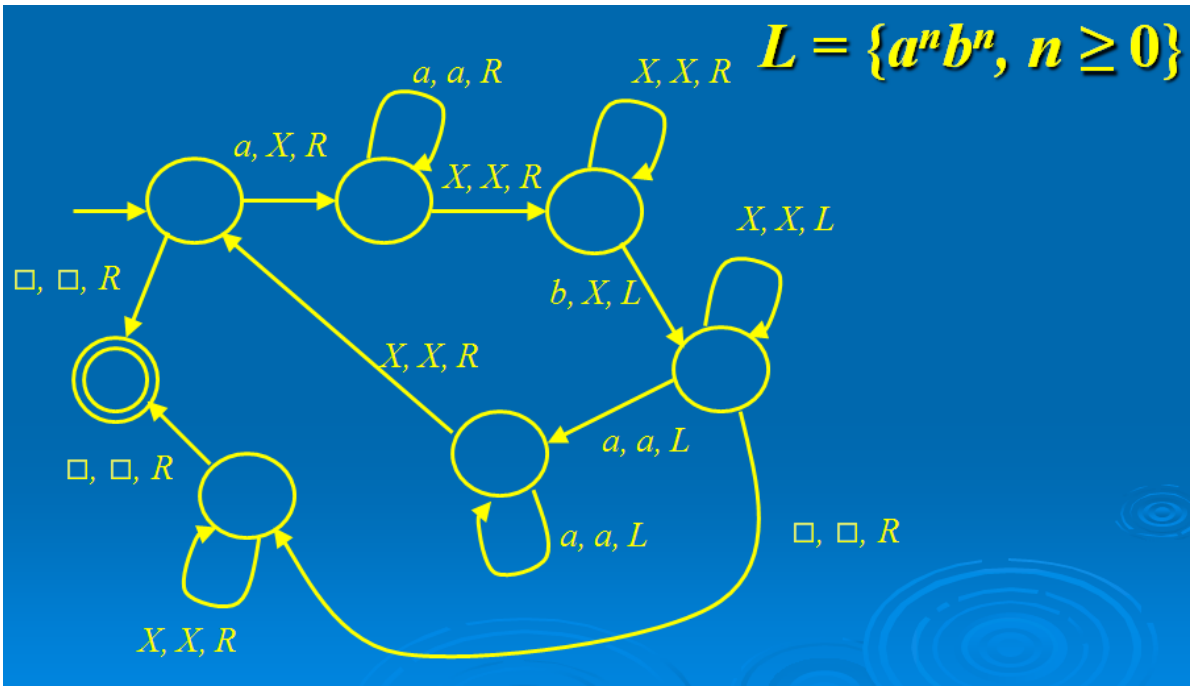
Let's mark off each ***a*** and matching ***b*** with an ***X***

Let's work with the ***a***'s and ***b***'s from left to right

Whenever we mark off an ***a***, move right until we find a ***b***. Mark off the ***b*** and head back left looking for the left-most remaining ***a***. Repeat as long as you can.

If there are no more ***a***'s, go to the right and check that there are no more ***b***'s.

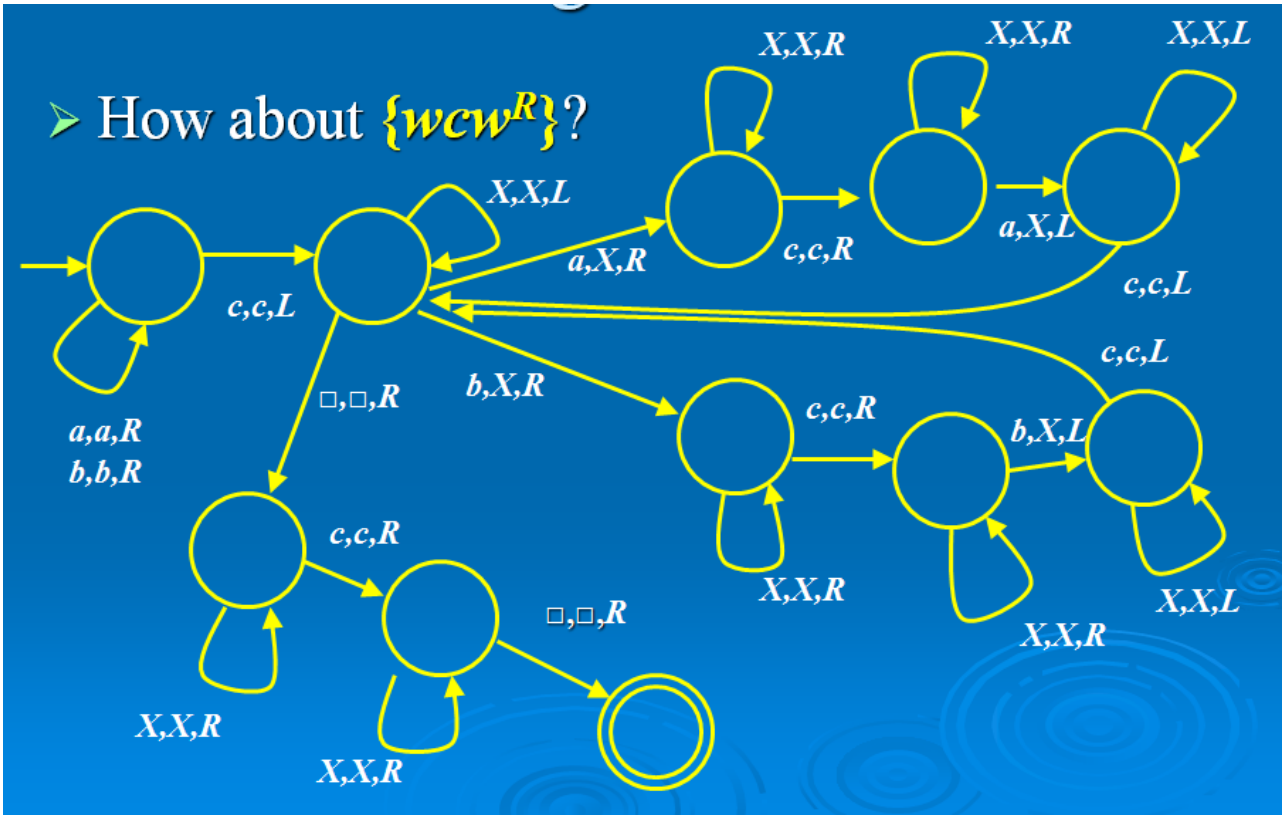
Let the machine die if there is no matching ***b*** for an ***a***.



Can it accept the language $\{wcw^R\}$? How?

Lay out a strategy

- Let's find the center ***c***.
- Find the next unmarked symbol on the left. Head right, remembering whether it was an ***a*** or a ***b***.
- If the next unmarked symbol on the right matches, head back to the middle and start again.
- If from the center ***c***, there are no more unmarked symbols on the left, make sure there are no more on the right as well.



How about $\{ww^R\}$?

Describe in words how a Turing machine can recognize this language.

Instead of going to the middle and working outwards like the last example, start with the outermost symbols and keep matching until you hit the middle.

How about $\{a^n b^n c^n\}$?

YES!!!!!!!!!!

consider our first machine and how it can be extended to handle matching ***c***'s as well.

Turing machines are more powerful than pushdown automata!

As you can see from these examples, while Turing machines are very powerful, they tend to get very complicated very quickly.

I would never ask you to construct in detail any Turing machine that requires more than a dozen states. In other cases, you may merely describe the algorithm used by such a Turing machine.

Transducers: Because Tm's can write to the tape, they can produce output rather than merely recognize certain types of input.

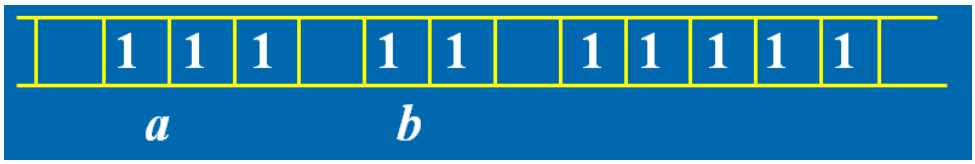
In our Tm world, we will represent numbers in the unary system: the value ***n*** will be represented as ***n*** 1's.

We can represent multiple numbers by putting a blank between them.

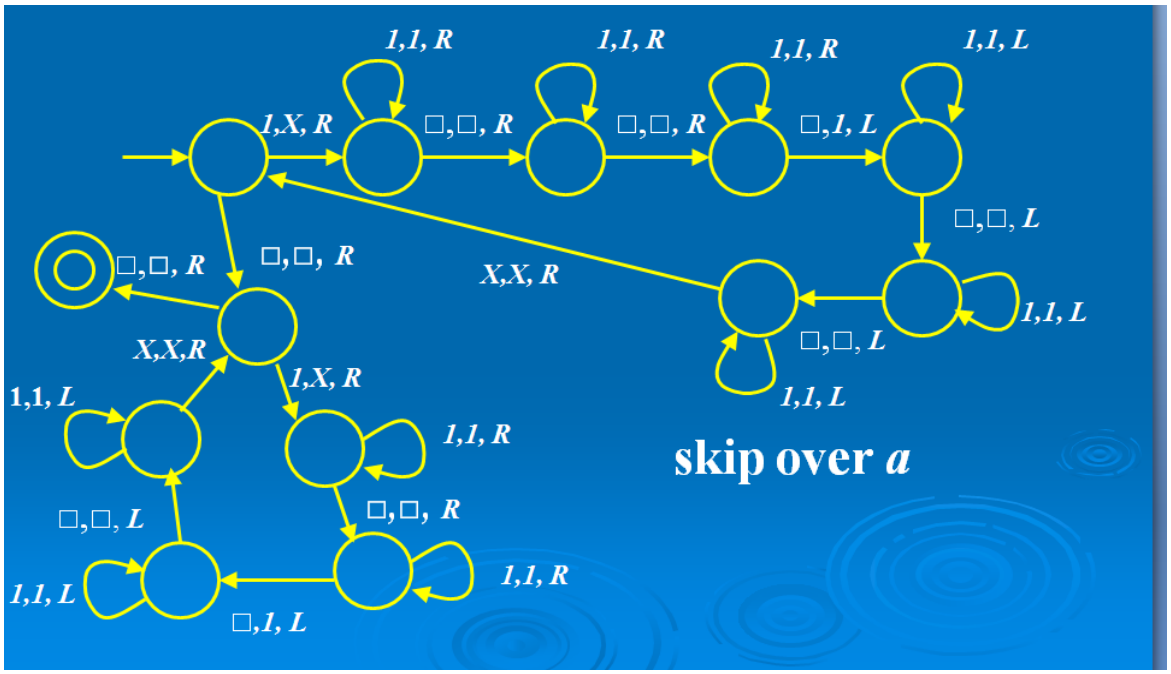
Transducers always start looking at the leftmost symbol of the leftmost "number."

We say we have a result if the machine stops in an accepting state. The "number" to the right of the tape head is the result.

How would a Turing machine compute addition?



Strategy: mark off each 1 in *a* and copy after the blank to the right of *b*. mark off each 1 in *b* and copy to the right of the 1's produced so far.



2. A variation on TM, TM encoding, Universal TM, Defining Computers by TM

3. Chomsky's hierarchy of grammars Turing machines theory

Chomsky Classification of Languages and Grammars:

According to Chomsky hierarchy, grammars are divided of 4 types:

Type 0 known as unrestricted grammar.

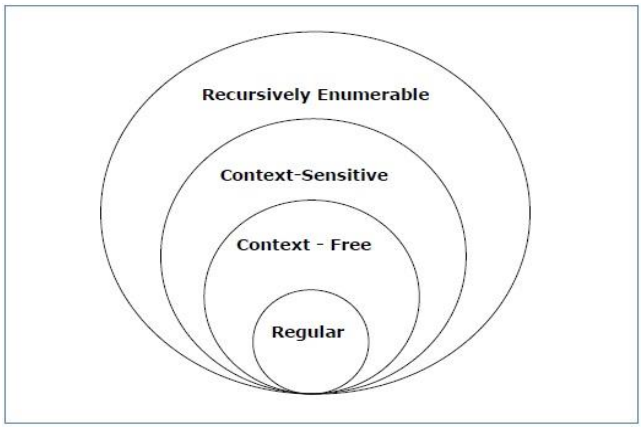
Type 1 known as context-sensitive grammar.

Type 2 known as context-free grammar.

Type 3 Regular Grammar.

Grammar Type	Production Rules	Language Accepted	Automata	Closed Under
Type-3 (Regular Gramar)	A⟶a or A⟶aB where A,B ∈ N(non terminal) and a∈T(Terminal)	Regular	Finite Automata	Union, Intersection, Complementation, Concatenation, Kleene Closure
Type-2 (Context Free Grammar)	A->ρ where A ∈N and ρ ∈ (TUN)*	Context Free	Push Down Automata	Union, Concatenation, Kleene Closure
Type-1 (Context Sensitive Grammar)	α⟶β where α, β∈ (TUN)* and len(α) <= len(β) and α should contain atleast 1 non terminal.	Context Sensitive	Linear Bound Automata	Union, Intersection, Complementation, Concatenation, Kleene Closure
Type-0 (Recursive Enumerable)	α ⟶ β where α, β∈ (TUN)* and α contains atleast 1 non-terminal	Recursive Enumerable	Turing Machine	Union, Intersection, Concatenation, Kleene Closure

The relationship between these can be represented as:



Type 0 (Unrestricted Grammar)

In Type 0: Type-0 grammars include all formal grammars. Type 0 grammar language is recognized by the Turing machine. These languages are also known as the recursively enumerable languages.

Grammar Production in the form of α➔β, where α is (V + T)* V (V + T)*

V : Variables

T : Terminals.

β is (V + T)*.

In type 0 there must be at least one variable on the Left side of production.

For example,

Sab → ba

A → S.

Here, Variables are S, A and Terminals a, b.

Type 1 (Context Sensitive)

Type-1 grammars generate the context-sensitive languages. The language generated by the grammar are recognized by the Linear Bound Automata

In Type 1

1. First of all Type 1 grammar should be Type 0.

2. Grammar Production in the form of

α→β

| α | <= | β |

i.e. the count of the symbol in α is less than or equal to β

For Example,

S → AB

AB → abc

B → b

Type 2 (Context Free)

Type-2 grammars generate the context-free languages. The language generated by the grammar is recognized by a Non-Deterministic Pushdown Automata. Type-2 grammars generate the context-free languages.

In Type 2,

1. First of all, it should be Type 1.

2. The left-hand side of a production can have only one variable.

| α | = 1.

There is no restriction on β.

For example,

S →AB

A → a

B → b

Type 3 (Regular Grammar)

Type-3 grammars generate the regular languages. These languages are exactly all languages that can be decided by a finite state automaton.

Type 3 is a most restricted form of grammar.

Type 3 should be in the given form only :

V → VT* / T*.

(or)

V → T*V /T*

for example :

S → ab.

Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X → a or X → aY**

where **X, Y ∈ N** (Nonterminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

```
X → ε
X → a | aY
Y → b
```

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form **A → γ**

where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

```
S → X a
```

X → a

X → aX

X → abc

X → ε

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

α A β → α γ β

where **A ∈ N** (Non-terminal)

and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals)

The strings **α** and **β** may be empty, but **γ** must be non-empty.

The rule **S → ε** is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

AB → AbBc

A → bcA

B → b

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of **α → β** where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

Example

S → ACaB

Bc → acB

CB → DB

aD → Db

4. Decidability

Decidable and Undecidable Problems:

A language is **Decidable or Recursive** if a Turing machine can be constructed which accepts the strings which are part of the language and rejects others. e.g.; A number is prime or not is a decidable problem.

A language is **Semi–Decidable or Recursive Enumerable** if a Turing machine can be constructed which accepts the strings which are part of the language and it may loop forever for strings which are not part of the language.

A problem is **undecidable** if we can't construct algorithms and Turing machine which can give yes or no answer. e.g.; Whether a CFG is ambiguous or not is undecidable.

Decidability Table						
Problem	RL	DCFL	CFL	CSL	Rl	REL
Membership Problem	D	D	D	D	D	UD
Empitnesss problem	D	D	D	UD	UD	UD
Completeness Problem	D	UD	UD	UD	UD	UD
Equality Problem	D	UD	UD	UD	UD	UD
Subset Problem	D	UD	UD	UD	UD	UD
L1L2 =	D	UD	UD	UD	UD	UD
Finiteness	D	D	D	UD	UD	UD
Complement is not of same type	D	D	UD	D	D	UD
Intersection is not same type	D	UD	UD	UD	UD	UD
Is l Regular	D	D	UD	UD	UD	UD

Decidability Table						
Problem	**RL**	**DCFL**	**CFL**	**CSL**	**RL**	**REL**
Membership Problem	D	D	D	D	D	UD
Emptiness Problem	D	D	D	UD	UD	UD
Completeness Problem	D	UD	UD	UD	UD	UD
Equality Problem	D	UD	UD	UD	UD	UD
Subset Problem	D	UD	UD	UD	UD	UD
$L1 \cap L2 = \phi$	D	UD	UD	UD	UD	UD
Finiteness	D	D	D	UD	UD	UD
Complement is of same type	D	D	UD	D	D	UD
Intersection is of same type	D	UD	UD	UD	UD	UD
Is L regular	D	D	UD	UD	UD	UD

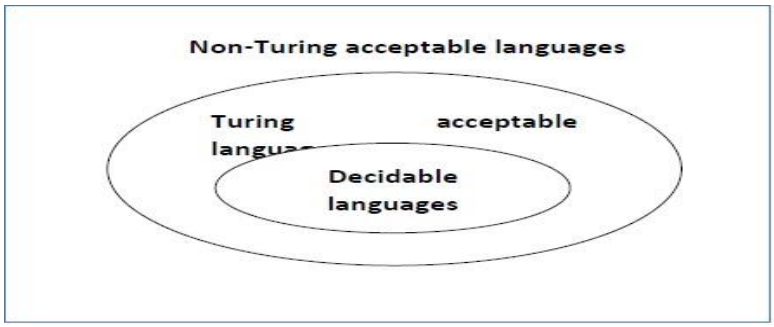
Countability:

- Set of all strings over any finite alphabet are countable.

- Every subset of countable set is either finite or countable.
- Set of all Turing Machines are countable.
- The set of all languages that are not recursive enumerable is Uncountable.

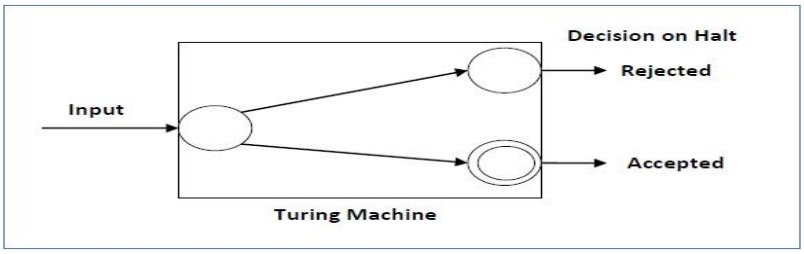
Language Decidability

A language is called **Decidable** or **Recursive** if there is a Turing machine which accepts and halts on every input string **w**. Every decidable language is Turing-Acceptable.



A decision problem **P** is decidable if the language **L** of all yes instances to **P** is decidable.

For a decidable language, for each input string, the TM halts either at the accept or the reject state as depicted in the following diagram −



Example 1

Find out whether the following problem is decidable or not −

Is a number ‘m’ prime?

Solution

Prime numbers = {2, 3, 5, 7, 11, 13, ……..}

Divide the number **‘m’** by all the numbers between ‘2’ and ‘√m’ starting from ‘2’.

If any of these numbers produce a remainder zero, then it goes to the “Rejected state”, otherwise it goes to the “Accepted state”. So, here the answer could be made by ‘Yes’ or ‘No’.

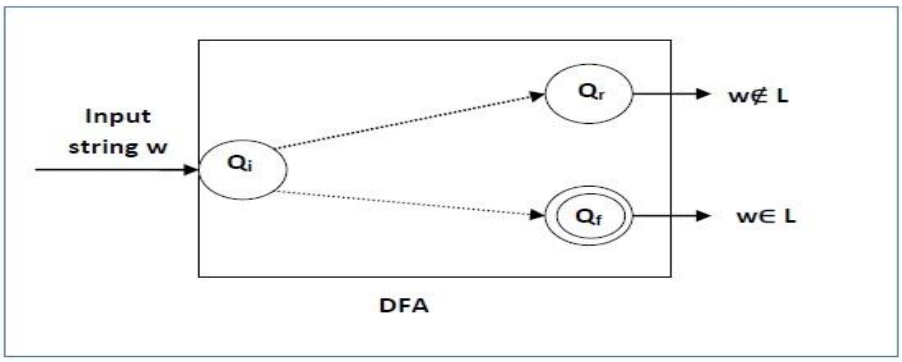
Hence, it is a decidable problem.

Example 2

Given a regular language **L** and string **w**, how can we check if **w ∈ L**?

Solution

Take the DFA that accepts **L** and check if **w** is accepted



Some more decidable problems are −

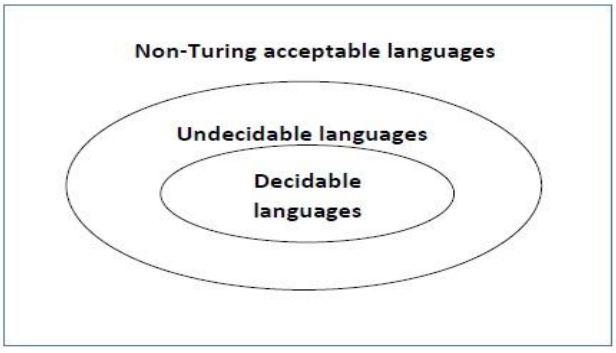
- Does DFA accept the empty language?
- Is $L_1 \cap L_2 = \emptyset$ for regular sets?

Note −

- If a language **L** is decidable, then its complement **L'** is also decidable
- If a language is decidable, then there is an enumerator for it.

Undecidable Languages

For an undecidable language, there is no Turing Machine which accepts the language and makes a decision for every input string **w** (TM can make decision for some input string though). A decision problem **P** is called "undecidable" if the language **L** of all yes instances to **P** is not decidable. Undecidable languages are not recursive languages, but sometimes, they may be recursively enumerable languages.



Example

- The halting problem of Turing machine
- The mortality problem
- The mortal matrix problem

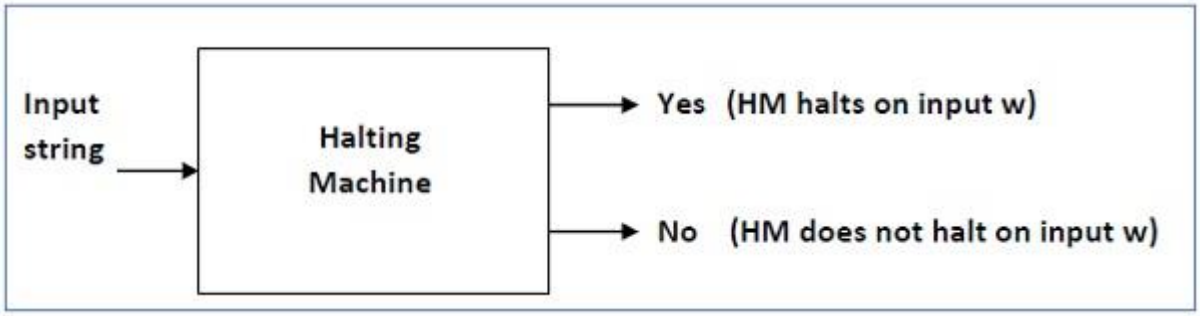
- The Post correspondence problem, etc.

Turing Machine Halting Problem

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

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

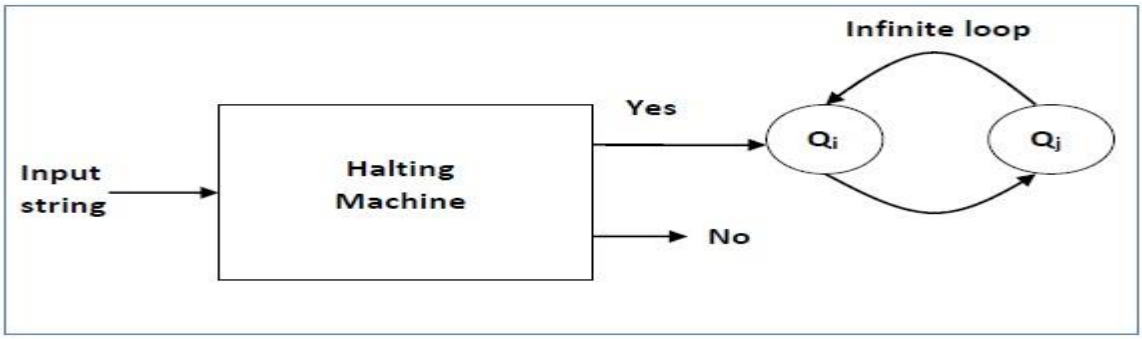
Proof − At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a **Halting machine** that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount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



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

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

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



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

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

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

Rice Theorem

Theorem: L = {<M> | L (M) ∈ P} is undecidable when p, a non-trivial property of the Turing machine, is undecidable.

If the following two properties hold, it is proved as undecidable −

Property 1 − If **M1** and **M2** recognize the same language, then either **<M1> <M2> ∈ L** or **<M1> <M2> ∉ L**

Property 2 − For some M1 and M2 such that **<M1> ∈ L** and **<M2> ∉ L**

Proof −

Let there are two Turing machines $\mathbf{X_1}$ and $\mathbf{X_2}$.

Let us assume $\mathbf{<X_1> \in L}$ such that

$\mathbf{L(X_1) = \varphi}$ and $\mathbf{<X_2> \notin L}$.

For an input 'w' in a particular instant, perform the following steps −

- If **X** accepts **w**, then simulate $\mathbf{X_2}$ on **x**.
- Run **Z** on input **<W>**.
- If **Z** accepts **<W>**, Reject it; and if **Z** rejects **<W>**, accept it.

If **X** accepts **w**, then

$L(W) = L(X_2)$ and $<W> \notin P$

If **M** does not accept w, then

$L(W) = L(X_1) = \varphi$ and $<W> \in P$

Here the contradiction arises. Hence, it is undecidable.

5. Post Machine

Post Correspondence Problem

The Post Correspondence Problem (PCP), introduced by Emil Post in 1946, is an undecidable decision problem. The PCP problem over an alphabet ∑ is stated as follows −

Given the following two lists, **M** and **N** of non-empty strings over ∑ −

$M = (x_1, x_2, x_3, \ldots\ldots, x_n)$

$N = (y_1, y_2, y_3, \ldots\ldots, y_n)$

We can say that there is a Post Correspondence Solution, if for some $i_1, i_2, \ldots\ldots i_k$, where $1 \le i_j \le n$, the condition $x_{i1} \ldots\ldots x_{ik} = y_{i1} \ldots\ldots y_{ik}$ satisfies.

Example 1

Find whether the lists M = (abb, aa, aaa) and N = (bba, aaa, aa) have a Post Correspondence Solution?

Solution

	x_1	x_2	x_3
M	Abb	aa	aaa
N	Bba	aaa	aa

Here, $x_2x_1x_3$ = 'aaabbaaa' and **$y_2y_1y_3$ = 'aaabbaaa'**

We can see that **$x_2x_1x_3 = y_2y_1y_3$**

Hence, the solution is **i = 2, j = 1, and k = 3.**

Example 2

Find whether the lists **M = (ab, bab, bbaaa)** and **N = (a, ba, bab)** have a Post Correspondence Solution?

Solution

	x_1	x_2	x_3
M	Ab	bab	bbaaa
N	A	ba	bab

In this case, there is no solution because, **$|x_2x_1x_3| \neq |y_2y_1y_3|$** (Lengths are not same).

Hence, it can be said that this Post Correspondence Problem is **undecidable**.