

स्वाध्याय

स्वमन्थन

स्वावलम्बन

UTTAR PRADESH RAJARSHI TANDON OPEN UNIVERSITY
(Established vide U.P. Govt. Act No. 10, of 1999)

BCA-18
THEORY OF COMPUTATION

FIRST BLOCK

Finite Automata and formal Languages



ra Gandhi National Open University



UP Rajarshi Tandon Open University

hantipuram (Sector-F), Phaphamau, Allahabad - 211013



Uttar Pradesh
Rajarshi Tandon Open University

BCA-18

**THEORY OF
COMPUTATION**

Block

1

FINITE AUTOMATA AND FORMAL LANGUAGES

UNIT 1

Finite Automata and Languages 7

UNIT 2

Non-deterministic Finite Automata 25

UNIT 3

Context Free Grammar 53

COURSE INTRODUCTION

CS-73 is a one-semester introductory course in the Theory of Computation in which the following topics are covered:

Formal languages
Automata theory
Recursive function theory
Uncomputability
Computational complexity and
Applications to solve practical problems.

The treatment of the subject matter is mathematical but the viewpoint is that of computer science. Care has been taken to exhibit the relationship between theoretical topics being presented and the applied topics with which you are expected to be familiar at this level.

In **Block 1** of this course we will expose you to the elementary approach to the study of languages through the concepts of alphabet, string, finite automata, grammar and classification of languages. We start with finite automata and formal languages and elaborate on some of their properties. Then we show that both the approaches are equivalent. Further, we generalize our classes of finite automata and formal languages.

You will find that some properties of finite automata and formal languages have very useful applications in compilers, verifying protocols and in description/specification of major parts of high-level programming languages and document-description languages that are patterned on context free grammars. The Hierarchy of automata theory and formal languages is developed in the context of constructing well-designed parser routines for a compiler.

In **Block 2** we continue our study of extensions of finite automata through detailed study of Turing Machine, named so in honour of the inventor of this model of computation, viz Alan Turing (1912-1954). Turing Machine is hitherto known ultimate formal model of computation.

Also in this block, we introduce another formal approach to computation, viz, recursive function approach. Our discussion leads us to the study of partial recursive functions which have computational power equivalent to that of Turing Machines.

In **Block 3** of this course, we focus on uncomputability, complexity and applications of the theories discussed in the course, to solve practical problems. Under uncomputability, we discuss a number of problems which are not solvable by the formal computational techniques known so far. For such problems or equivalent languages, we will introduce you to standard forms of such problems, and also discuss their properties. Also, for the problems which are solvable, i.e., for the problems whose language is decidable, we discuss quantitative classification of decidable languages by considering Turing Machines that are restricted, not in their structural capabilities, but in the amount of effort they are allowed to expend when computing on an input string.

Open classification problems relating to the classes P and NP also form part of our discussion of decidable languages.

Now, a word about the way we have presented this course. In each of the three blocks we first make a general introduction to the block. Then we present the detailed contents of the units of the block. In each unit you will find plenty of exercises interspersed within the text. Please try the exercises as and when you come across

these. They are meant to help you check whether you've understood the material that is being discussed. We have also given our solutions to the exercises in a section at the end of the unit.

While you go through the course, you will notice that each unit is divided into sections. These sections are often further divided into sub-sections. The sections/sub-sections of a unit are numbered sequentially, as are the exercises, theorems and important equations in it. Since the material in the different units are heavily interlinked, cross-references are quite frequent. For this purpose we use the notation $\text{Sec. } x.y$ to mean Section y of Unit x .

Another compulsory component of this course is an assignment, which you should attempt after studying all the blocks of the course. Your counselor will evaluate and return it to you with detailed comments. Thus, the assignment is a teaching as well as an assessment aid.

The course material that we have sent you is self-sufficient. If you have any problem in understanding any portion of it, please ask your counsellor for help. Also, if you feel like studying any topic in greater depth, you may consult:

- 1.) "Introduction to Automata Theory, Languages, and Computation," by John Hopcroft, Rajeev Motwani, Jeffery D. Ullman, Pearson Education (2001).
- 2.) "Elements of the Theory of Computation" by Lewis, Papadimitriou, Prentice Hall of India (1981).

These books will be available at your study center.

We hope you will enjoy this course!

BLOCK INTRODUCTION

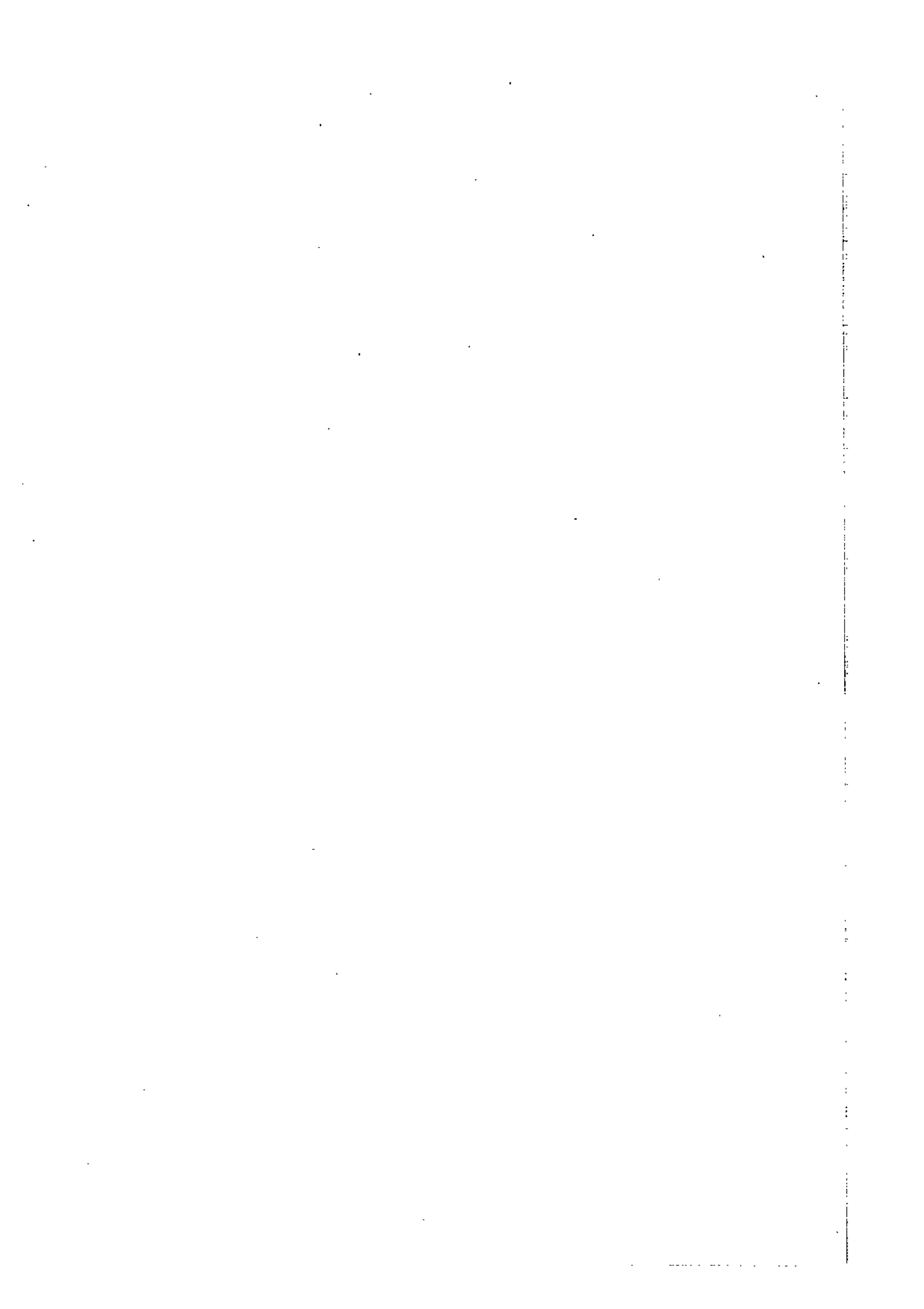
In Block I, we shall discuss the general theory of automata, regular expressions and their properties, Languages and Grammar. Besides, sufficient attention is devoted to such topics as equivalence of regular expression and finite automata and equivalence of pushdown automata and context-free languages. The theory of computation arose in the 50's when computer scientists were trying to use computers to translate one language into another. Now the theories of computation, formal language, and automata, which have emerged as mathematical models of programming languages and computers have a wide range of application in computing techniques. All of these theoretical developments bear directly on what computer scientists do today.

In Unit 1, we begin with a view of what automata theory is about and what its uses are. A section introduces strings, language, and regular expression. We shall discuss few machines based on output namely moore and mealy machines.

In Unit 2, we shall discuss one special and commonly used automaton, which are non-deterministic finite automata. We shall derive an equivalence in between two approaches i.e. a language can be derived from a finite automata as well as from a regular expression. Also, in this unit, we shall also introduce Pumping lemma to obtain whether a language is regular or not.

In Unit 3, which is the last unit of this block, we shall discuss grammar and its classification. We shall discuss the context free languages in detail and push down automata. The equivalence of pushdown automata and context free language is also discussed in the unit. We shall also discuss pumping lemma for a context free language.

Now, a few suggestions that may help you study the units in this block. Do try the exercises in the units in this block as and when you come to them. This will help you to confirm your understanding of the related study material. After finishing this block please try the assignment questions which are based on this block.



UNIT 1 FINITE AUTOMATA AND LANGUAGES

Structure	Page Nos.
1.0 Introduction	7
1.1 Objectives	7
1.2 Regular Expressions	7
1.2.1 Introduction to Defining of Languages	
1.2.2 Kleene closure definition	
1.2.3 Formal Definition of Regular Expressions	
1.2.4 Algebra of Regular Expressions	
1.3 Regular Languages	13
1.4 Finite Automata	14
1.4.1 Finite Automata	
1.4.2 Another method to describe FA	
1.4.3 Finite Automata as Output Devices	
1.5 Summary	23
1.6 Solutions/Answers	23

1.0 INTRODUCTION

We shall study different types of theoretical machines that are mathematical models for actual physical processes. By considering the possible inputs on which these machines can work, we can analyze their various strengths and weaknesses. We then arrive at what we may believe to be the most powerful machine possible. When we do so, we would be surprised to find the computational tasks that this machine cannot perform. This will be our ultimate result that no matter what machine we build, there will always be questions that are simple to state but even the most powerful machine possibly cannot answer. Along the way, we hope you would understand the concept of computability, which is the foundation of further research in this field.

1.1 OBJECTIVES

After studying this unit, you should be able to:

- define alphabet, substring;
 - define a language and various operations on languages;
 - define and use a regular expression;
 - define a finite automata for computation of a language; and
 - obtain a finite automata for a known language;
-

1.2 REGULAR EXPRESSIONS

In this unit, first we shall discuss the definitions of alphabet, string, and language with some important properties.

1.2.1 Introduction to Defining of Languages

For a language, defining rules can be of two types. The rules can either tell us how to test a string of alphabet letters that we might be presented with, to see if it is a valid word, i.e., a word in the language or the rules can tell us how to construct all the words in the language by some clear procedures.

Alphabet: A finite set of symbols/characters. We generally denote an alphabet by Σ . If we start an alphabet having only one letter, say, the letter z , then $\Sigma = \{z\}$

Letter : Each symbol of an alphabet may also be called a letter of the alphabet or simply a letter.

Language over an alphabet : A set of words over an alphabet. Languages are denoted by letter L with or without a subscript.

String/word over an alphabet: Every member of any language is said to be a string or a word.

Example 1: Let L_1 be the language of all possible strings obtained by
 $L_1 = \{z, zz, zzz, zzzz, \dots\}$

This can also be written as
 $L_1 = \{z^n\}$ for $n = 1, 2, 3, \dots$

A string of length zero is said to be **null string** and is represented by \wedge . Above given language L_1 does not include the null string. We could have defined it so as to include \wedge . Thus, $L = \{z^n \mid n=0, 1, 2, 3, \dots\}$ contains the null string.

In this language, as in any other, we can define the operation of concatenation, in which two strings are written down side by side to form a new longer string. Suppose $u = ab$ and $v = baa$, then uv is called concatenation of two strings u and v and is $uv = abbaa$ and $vu = baaab$. The words in this language clearly analogous to the positive integers, and the operation of concatenation are analogous to addition:

z^n concatenated with z^m is the word z^{n+m} .

Example 2: If the word zzz is called c and the word zz is called d , then the word formed by concatenating c and d is
 $cd = zzzzz$

When two words in our language L_1 are concatenated they produce another word in the language L_1 . However, this may not be true in all languages.

Example 3: If the language is $L_2 = \{z, zzz, zzzzz, zzzzzzz, \dots\}$
 $= \{z^{\text{odd}}\}$
 $= \{z^{2n+1} \text{ for } n = 0, 1, 2, 3, \dots\}$

then $c = zzz$ and $d = zzzzz$ are both words in L_2 , but their concatenation $cd = zzzzzzzz$ is not a word in L_2 . The reason is simple that member of L_2 are of odd length while after concatenation it is of even length.

Note: The alphabet for L_2 is the same as the alphabet for L_1 .

Example 4: A Language L_3 may denote the language having strings of even lengths include of length 0. In other words, $L_3 = \{\wedge, zz, zzzz, \dots\}$

Another interesting language over the alphabet $\Sigma = \{z\}$ may be

Example 5: $L_4 = \{z^p \mid p \text{ is a prime natural number}\}$.
There are infinitely many possible languages even for a single letter alphabet $\Sigma = \{z\}$.

In the above description of concatenation we find very commonly, that for a single letter alphabet when we concatenate c with d , we get the same word as when we

concatenate d with c , that is $cd = dc$ But this relationship does not hold for all languages. For example, in the English language when we concatenate "Ram" and "goes" we get "Ram goes". This is, indeed, a word but distinct from "goes Ram".

Now, let us define the reverse of a language L . If c is a word in L , then reverse (c) is the same string of letters spelled backward.

The reverse (L) = {reverse (w), $w \in L$ }

Example 6: Reverse (zzz) = zzz
Reverse (173) = 371

Let us define a new language called PALINDROME over the alphabet $\Sigma = \{a,b\}$.

PALINDROME = { Λ , and all strings w such that reverse (w) = w }
= { Λ , a , b , aa , bb , aaa , aba , bab , bbb , $aaaa$, $abba$, ...}

Concatenating two words in PALINDROME may or may not give a word in palindrome, e.g., if $u = abba$ and $v = abbcba$, then $uv = abbaabbcba$ which is not palindrome.

1.2.2 Kleene Closure Definition

Suppose an alphabet Σ , and define a language in which any string of letters from Σ is a word, even the null string. We shall call this language the closure of the alphabet. We denote it by writing $*$ after the name of the alphabet as a superscript, which is written as Σ^* . This notation is sometimes also known as Kleene Star.

For a given alphabet Σ , the language L consists of all possible strings, including the null string.

For example, if $\Sigma = \{z\}$, then, $\Sigma^* = L_1 = \{\Lambda, z, zz, zzz, \dots\}$

Example 7: If $\Sigma = \{0, 1\}$, then, $\Sigma^* = \{\Lambda, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$
So, we can say that Kleene Star is an operation that makes an infinite language of strings of letters out of an alphabet, if the alphabet, $\Sigma \neq \phi$. However, by the definition alphabet Σ may also be ϕ . In that case, Σ^* is finite. By "infinite language, we mean a language with infinitely many words.

Now, we can generalise the use of the star operator to languages, i.e., to a set of words, not just sets of alphabet letters.

Definition: If s is a set of words, then by s^* we mean the set of all finite strings formed by concatenating words from s , where any word may be used as often.

Example 8: If $s = \{cc, d\}$, then
 $s^* = \{\Lambda$ or any word composed of factors of cc and $d\}$
 $= \{\Lambda$ or all strings of c 's and d 's in which c 's occur in even clumps}.

The string $ccdcc$ is not in s^* since it has a clump of c 's of length 3.
 $\{x : x = \Lambda$ or $x = (cc)^{i_1} d^{j_1} (cc)^{i_2} d^{j_2} \dots (cc)^{i_m} (d)^{j_m}\}$ where $i_1, j_1, \dots, i_m, j_m \geq 0$

Positive Closure: If we want to modify the concept of closure to refer to only the concatenation leading to non-null strings from a set s , we use the notation $+$ instead of $*$. This plus operation is called positive closure.

Theorem 1: For any set s of strings prove that $s^+ = (s^*)^+ = s^{**}$

Proof: We know that every word in s^{**} is made up of factors from s^* .
Also, every factor from s^* is made up of factors from s .
Therefore, we can say that every word in s^{**} is made up of factors from s .

First, we show $s^{**} \subset s^*$. (i)

Let $x \in s^{**}$. Then $x = x_1 \dots x_n$ for some $x_i \in s^*$ which implies $s^{**} \subset s^*$.

Next, we show $s^* \subset s^{**}$.

$s^* \subset s^{**}$ (ii)

By above inclusions (i) and (ii), we prove that
 $s^* = s^{**}$

Now, try some exercises.

Ex.1) If $u = ababb$ and $v = baa$ then find
(i) uv (ii) vu (iii) uu (iv) vv (v) uuv .

Ex.2) Write the Kleene closure of the following

- (i) $\{aa, b\}$
(ii) $\{a, ba\}$

1.2.3 Formal Definition of Regular Expressions

Certain sets of strings or languages can be represented in algebraic fashion, then these algebraic expressions of languages are called **regular expressions**. Regular expressions are in **Bold face**. The symbols that appear in regular use of the letters of the alphabet Σ are the symbol for the null string \wedge , parenthesis, the star operator, and the plus sign.

The set of regular expressions is defined by the following rules:

1. Every letter of Σ can be made into a regular expression \wedge itself is a regular expression.
2. If l and m are regular expressions, then so are
 - (i) (l)
 - (ii) lm
 - (iii) $l+m$
 - (iv) l^*
 - (v) $l^+ = ll^*$
3. Nothing else is regular expression.

For example, now we would build expression from the symbols 0,1 using the operations of union, concatenation, and Kleene closure.

- (i) **01** means a zero followed by a one (concatenation)
(ii) **0+1** means either a zero or a one (union)
(iii) **0*** means $\wedge+0+00+000+\dots$ (Kleen closure).

With parentheses, we can build larger expressions. And, we can associate meanings with our expressions. Here's how

Expression	Set represented
$(0+1)^*$	all strings over $\{0,1\}$
$0^*10^*10^*$	strings containing exactly two ones
$(0+1)^*11$	strings which end with two ones.

The language denoted/represented by the regular expression R is $L(R)$.

Example 9: The language L defined by the regular expression ab^*a is the set of all strings of a's and b's that begin and end with a's, and that have nothing but b's inside.

$$L = \{aa, aba, abba, abbb, ab^4b, \dots\}$$

Example 10: The language associated with the regular expression a^*b^* contains all the strings of a's and b's in which all the a's (if any) come before all the b's (if any).

$$L = \{\Lambda, a, b, aa, ab, bb, aaa, aab, abb, bbb, aaa, \dots\}$$

Note that ba and aba are not in this language. Notice also that there need not be the same number of a's and b's.

Example 11: Let us consider the language L defined by the regular expression $(a+b)^*a(a+b)^*$. The strings of the language L are obtained by concatenating a string from the language corresponding to $(a+b)^*$ followed by a followed by a string from the language associated with $(a+b)^*$. We can also say that the language is a set of all words over the alphabet $\Sigma = \{a,b\}$ that have an a in them somewhere.

To make the association/correspondence/relation between the regular expressions and their associated languages more explicit, we need to define the operation of multiplication of set of words.

Definition: If S and T are sets of strings of letters (they may be finite or infinite sets), we define the product set of strings of letters to be $ST = \{\text{all combinations of a string from S concatenated with a string from T in that order}\}$.

Example 12: If $S = \{a, aa, aaa\}$, $T = \{bb, bbb\}$
Then, $ST = \{abb, abbb, aabb, aabbb, aaabb, aaabbb\}$.

Example 13: If $S = \{a, bb, bab\}$, $T = \{\Lambda, bbbb\}$
Then, $ST = \{a, bb, bab, abbbb, bbbbbb, babbbbb\}$

Example 14: If L is any language, Then, $L\Lambda = \Lambda L = L$.

Ex.3) Find a regular expression to describe each of the following languages:

- (a) $\{a,b,c\}$
- (b) $\{a,b,ab,ba,abb,baa,\dots\}$
- (c) $\{\Lambda,a,abb,abbbb,\dots\}$

Ex.4) Find a regular expression over the alphabet $\{0,1\}$ to describe the set of all binary numerals without leading zeroes (except 0 itself). So the language is the set

$$\{0,1,10,11,100,101,110,111,\dots\}.$$

1.2.4 Algebra of Regular Expressions

There are many general equalities for regular expressions. We will list a few simple equalities together with some that are not so simple. All the properties can be verified by using properties of languages and sets. We will assume that R,S and T denote the arbitrary regular expressions.

Properties of Regular Expressions

1. $(R+S)+T = R+(S+T)$
2. $R+R = R$

3. $R + \phi = \phi + R = R$.
4. $R + S = S + R$
5. $R\phi = \phi R = \phi$
6. $R\wedge = \wedge R = R$
7. $(RS)T = R(ST)$
8. $R(S+T) = RS+RT$
9. $(S+T)R = SR+TR$
10. $\phi^* = \wedge^* = \wedge$
11. $R^*R^* = R^* = (R^*)^*$
12. $RR^* = R^*R = R^* = \wedge + RR^*$
13. $(R+S)^* = (R^*S^*)^* = (R^*+S^*)^* = R^*S^* = (R^*S)^*R^* = R^*(SR^*)^*$
14. $(RS)^* = (R^*S^*)^* = (R^*+S^*)^*$

Theorem 2: Prove that $R+R = R$

Proof: We know the following equalities:

$$L(R+R) = L(R)UL(R) = L(R)$$

$$\text{So } R+R = R$$

Theorem 3: Prove the distributive property

$$R(S+T) = RS+RT$$

Proof: The following set of equalities will prove the property:

$$\begin{aligned} L(R(S+T)) &= L(R)L(S+T) \\ &= L(R)(L(S)UL(T)) \\ &= (L(R)L(S))U(L(R)L(T)) \\ &= L(RS+RT) \end{aligned}$$

Similarly, by using the equalities we can prove the rest. The proofs of the rest of the equalities are left as exercises.

Example 15: Show that $R+RS^*S = a^*bS^*$, where $R = b+aa^*b$ and S is any regular expression.

$$\begin{aligned} R+RS^*S &= R\wedge+RS^*S \text{ (property 6)} \\ &= R(\wedge+S^*S) \text{ (property 8)} \\ &= R(\wedge+SS^*) \text{ (property 12)} \\ &= RS^* \text{ (property 12)} \end{aligned}$$

$$\begin{aligned}
 &= (b+aa^*b)^*S^* \text{ (definition of R)} \\
 &= (\wedge+aa^*)^*bS^* \text{ (properties 6 and 8)} \\
 &= a^*bS^*. \text{ (Property 12)}
 \end{aligned}$$

Try an exercise now.

Ex.5) Establish the following equality of regular expressions:
 $b^*(abb^*+aabb^*+aaabb^*)^* = (b+ab+aab+aaab)^*$

As we already know the concept of language and regular expressions, we have an important type of language derived from the regular expression, called **regular language**.

1.3 REGULAR LANGUAGES

Language represented by a regular expression is called a regular language. In other words, we can say that a regular language is a language that can be represented by a regular expression.

Definition: For a given alphabet Σ , the following rules define the regular language associated with a regular expression.

Rule 1: ϕ , $\{\wedge\}$ and $\{a\}$ are regular languages denoted respectively by regular expressions ϕ and \wedge .

Rule 2: For each a in Σ , the set $\{a\}$ is a regular language denoted by the regular expression a .

Rule 3: If l is a regular expression associated with the language L and m is a regular expression associated with the language M , then:

- (i) The language $\{xy : x \in L \text{ and } y \in M\}$ is a regular expression associated with the regular expression lm
- (ii) The regular expression $l+m$ is associated with the language formed by the union of the sets L and M .

$$\text{language } (l+m) = L \cup M$$

- (iii) The language associated with the regular expression $(l)^*$ is L^* , the Kleen Closure of the set L as a set of words:

$$\text{language } (l^*) = L^*$$

Now, we shall derive an important relation that, all finite languages are regular.

Theorem 4: If L is a finite language, then L can be defined by a regular expression. In other words, all finite languages are regular.

Proof: A language is finite if it contains only finitely many words.

To make one regular expression that defines the language L , turn all the words in L into bold face type and insert plus signs between them. For example, the regular expression that defines the language $L = \{\text{baa}, \text{abbba}, \text{bababa}\}$ is **baa + abbba + bababa**

Example 6: If $L = \{aa, ab, ba, bb\}$, then the corresponding regular expression is $aa + ab + ba + bb$.

Another regular expression that defines this language is $(a+b)(a+b)$.

So, a particular regular language can be represented by more than one regular expressions. Also, by definition, each regular language must have at least one regular expression corresponding to it.

Try some exercises.

Ex.6) Find a language to describe each of the following regular expressions:

(a) $a+b$ (b) $a+b^*$ (c) a^*bc^*+ac

Ex.7) Find a regular expression for each of the following languages over the alphabet $\{a,b\}$:

(a) strings with even length.
(b) strings containing the sub string aba.

In our day to day life we oftenly use the word Automatic. Automation is the process where the output is produced directly from the input without direct involvement of mankind. The input passes from various states in process for the processing of a language we use very important finite state machine called finite automata.

1.4 FINITE AUTOMATA

Finite automata are important in science, mathematics, and engineering. Engineers like them because they are superb models for circuits (and, since the advent of VLSI systems sometimes finite automata represent circuits.) computer scientists adore them because they adapt very likely to algorithm design. For example, the lexical analysis portion of compiling and translation. Mathematicians are introduced by them too due to the fact that there are several nifty mathematical characterizations of the sets they accept.

Can a machine recognise a language? The answer is yes for some machine and some an elementary class of machines called finite automata. Regular languages can be represented by certain kinds of algebraic expressions by Finite automaton and by certain grammars. For example, suppose we need to compute with numbers that are represented in scientific notation. Can we write an algorithm to recognise strings of symbols represented in this way? To do this, we need to discuss some basic computing machines called finite automaton.

An automata will be a finite automata if it accepts all the words of any regular language where language means a set of strings. In other words, The class of regular language is exactly the same as the class of languages accepted by FA's., a deterministic finite automata.

1.4.1 Finite Automata

A system where energy and information are transformed and used for performing some functions without direct involvement of man is called automaton. Examples are automatic machine tools, automatic photo printing tools, etc.

A finite automata is similar to a finite state machine. A finite automata consists of five parts:

(1) a finite set of states;

- (2) a finite set of alphabets;
- (3) an initial state;
- (4) a subset of set of states (whose elements are called "yes" state or; accepting state;) and
- (5) a next-state function or a transition state function.

A finite automata over a finite alphabet A can be thought of as a finite directed graph with the property that each node emits one labelled edge for each distinct element of A . The nodes are called states. There is one special state called the start (or initial) state, and there is a possible empty set of states called final states.

For example, the labelled graph in fig.1 given below represents a DFA over the alphabet $A = \{a,b\}$ with start state 1 and final state 4.

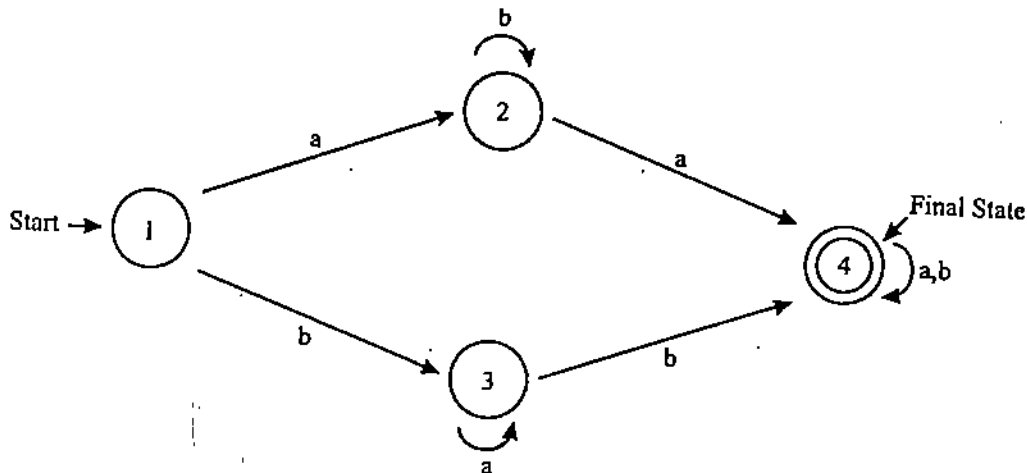


Fig. 1: Finite Automata

We always indicate the start state by writing the word start with an arrow pointing to it. Final states are indicated by double circle.

The single arrow out of state 4 labelled with a,b is short hand for two arrows from state 4, going to the same place, one labelled a and one labelled b. It is easy to check that this digraph represents a DFA over $\{a,b\}$ because there is a start state, and each state emits exactly two arrows, one labelled with a and one labelled with b.

So, we can say that a finite automaton is a collection of three tuples:

1. A finite set of states, one of which is designated as the initial state, called the start state, and some (may be none) of which we designated as final states.
2. An alphabet Σ of possible input letters from which are formed strings that are to be read one letter at a time.
3. A finite set of transitions that tell for each state and for each letter of the input alphabet which state to go to next.

For example the input alphabet has only two letters a and b. Let us also assume that there are only three states, x, y and z. Let the following be the rules of transition:

1. from state x and input a go to state y;
2. from state x and input b go to state z;
3. from state y and input b go to state x;

4. from state y and input b go to state z; and

5. from state z and any input stay at state z.

Let us also designate state x as the starting state and state z as the only final state.

Let us examine what happens to various input strings when presented to this FA. Let us start with the string aaa. We begin, as always, in state x. The first letter of the string is an a, and it tells us to go state y (by rule 1). The next input (instruction) is also an a, and this tells us (by rule 3) to go back to state x. The third input is another a, and (by Rule 1) again we go to the state y. There are no more input letters in the input string, so our trip has ended. We did not finish in the final state (state z), so we have an unsuccessful termination of our run.

The string aaa is not in the language of all strings that leave this FA in state z. The set of all strings that do leave as in a final state is called the language defined by the finite automaton. The input string aaa is not in the language defined by this FA. We may say that the string aaa is not accepted by this FA because it does not lead to a final state. We may also say "aaa is rejected by this FA." The set of all strings accepted is the language associated with the FA. So, we say that L is the language accepted by this FA. FA is also called a language recogniser.

Let us examine a different input string for this same FA. Let the input be abba. As always, we start in state x. Rule 1 tells us that the first input letter, a, takes us to state y. Once we are in state y we read the second input letter, which is b. Rule 4 now tells us to move to state z. The third input letter is a b, and since we are in state z, Rule 5 tells us to stay there. The fourth input letter is an a, and again Rule 5 says state z. Therefore, after we have followed the instruction of each input letter we end up in state z. State z is designated as a final state. So, the input string abba has taken us successfully to the final state. The string abba is therefore a word in the language associated with this FA. The word abba is accepted by this FA.

It is not difficult for us to predict which strings will be accepted by this FA. If an input string is made up of only the letter a repeated some number of times, then the action of the FA will be jump back and forth between state x and state y. No such word can ever be accepted.

To get into state z, it is necessary for the string to have the letter b in it as soon as a b is encountered in the input string, the FA jumps immediately to state z no matter what state it was before. Once in state z, it is impossible to leave. When the input strings run out, the FA will still be in state z, leading to acceptance of the string.

So, the FA above will accept all the strings that have the letter b in them and no other strings. Therefore, the language associated with this FA is the one defined by the regular expression $(a+b)^* b(a+b)^*$.

The list of transition rules can grow very long. It is much simpler to summarise them in a table format. Each row of the table is the name of one of the states in FA, and each column of this table is a letter of the input alphabet. The entries inside the table are the new states that the FA moves into the transition states. The transition table for the FA we have described is:

Table 1

State	Input	
	a	b
Start x	y	z
y	x	z
Final z	z	z

The machine we have already defined by the transition list and the transition table can be depicted by the state graph in Figure 2.

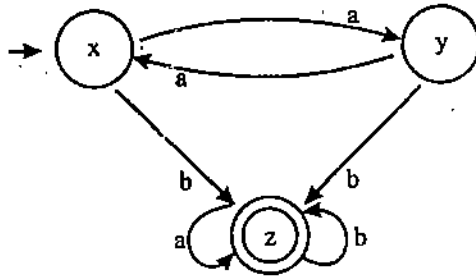


Fig. 2: State Transition graph

Note: A single state can be start as well as final state both. There will be only one start state and none or more than one final states in Finite Automaton.

1.4.2 Another Method to Describe FA

There is a traditional method to describe finite automata which is extremely intuitive. It is a picture called a graph. The states of the finite automaton appear as vertices of the graph while the transitions from state to state under inputs are the graph edges. The state graph for the same machine also appears in Figure3 given below.

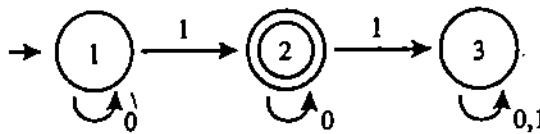


Fig. 3: Finite automata

The finite automata shown in Figure 3 can also be represented in Tabular form as below:

Table 2

	State	Input		Accept?
		0	1	
Start	1	1	2	No
Final	2	2	3	Yes
	3	3	3	No

Before continuing, let's examine the computation of a finite automaton. Our first example begins in state one and reads the input symbols in turn changing states as necessary. Thus, a computation can be characterized by a sequence of states. (Recall that Turing machine configurations needed the state plus the tape content. Since a finite automaton never writes, we always know what is on the tape and need only look at a state as a configuration.) Here is the sequence for the input 0001001.

Input Read : 0 0 0 1 0 0 1
 States : 1 → 1 → 1 → 1 → 2 → 2 → 2 → 3

Example 17 (An elevator controller): Let's imagine an elevator that serves two floors. Inputs are calls to a floor either from inside the elevator or from the floor itself. This makes three distinct inputs possible, namely:

- 0 - no calls
- 1 - call to floor one
- 2 - call to floor two

The elevator itself can be going up, going down, or halted at a floor. If it is on a floor, it could be waiting for a call or about to go to the other floor. This provides us with the six states shown in figure 4 along with the state graph for the elevator controller.

- W1 Waiting on first floor
- U1 About to go up
- UP Going up
- DN Going down
- W2 Waiting-second floor
- D2 About to go down

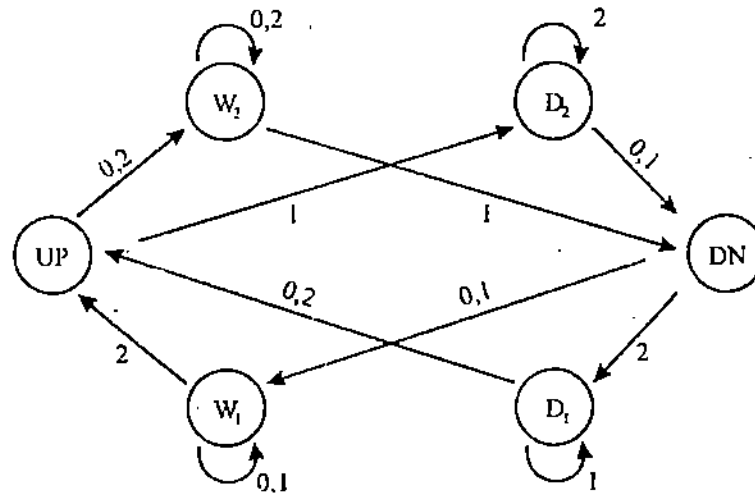


Fig. 4: Elevator Control

A transition state table for the elevator is given in Table 3.

Table 3: Elevator Control

State	Input		
	None	call to 1	call to 2
W1 (wait on 1)	W1	U1	UP
U1 (start up)	UP	U1	UP
UP	W2	D2	W2
DN	W1	W1	U1
W2 (wait on 2)	W2	DN	W2
D2 (start down)	DN	DN	D2

Accepting and rejecting states are not included in the elevator design because acceptance is not an issue. If we were to design a more sophisticated elevator, it might have states that indicated:

Finite automata

- a) power failure
- b) overloading, or
- c) breakdown

In this case, acceptance and rejection might make sense.

Let us make a few small notes about the design. If the elevator is about to move (i.e., in state U1 or D2) and it is called to the floor it is presently on it will stay. (This may

be good. Try it next time you are in an elevator.) And, if it is moving (up or down) and gets called back the other way, it remembers the call by going to the U1 or D2 state upon arrival on the next floor. Of course, the elevator does not do things like open and close doors (these could be states too) since that would have added complexity to the design. Speaking of complexity, imagine having 100 floors. That is our levity for this section. Now that we know what a finite automaton is, we must (as usual) define it precisely

Definition : A finite automaton M is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$ where :

- Q is a finite set (of states)
- Σ is a finite alphabet (of input symbols)
- $\delta: Q \times \Sigma \rightarrow Q$ (next state function)
- $q_0 \in Q$ (the starting state)
- $F \subseteq Q$ (the accepting states)

We also need some additional notation. The next state function is called the transition function and the accepting states are often called final states. The entire machine is usually defined by presenting a transition state table or a transition diagram. In this way, the states, alphabet, transition function, and final states are constructively defined. The starting state is usually the lowest numbered state. Our first example of a finite automaton is:

$$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \delta, q_1, \{q_2\})$$

Where the transition function δ , is defined explicitly by either a state table or a state graph.

At this point, we must make a slight detour and examine a very important yet seemingly insignificant input string called the empty string. It is a string without any symbols in it and is denoted as ϵ . It is not a string of blanks. An example might make this clear. Look between the brackets in the picture below.

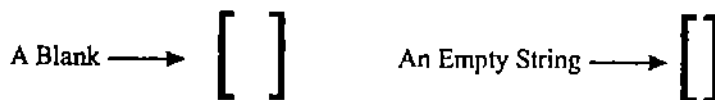


Fig. 5: Representation of a blank and an empty string

Let us look again at a computation by our first finite automaton. For the input 010, our machine begins in q_1 , reads a 0 and goes to $\delta(q_2, 0) = q_2$ after reading the final 0. All that can be put together as:

$$\delta(\delta(\delta(q_1, 0), 1)0) = q_2$$

We call this transition on strings δ^* and define it as follows:

Definition : Let $M = (Q, \Sigma, \delta, q_0, F)$. For any input string x , input symbol a , and state q_i , the transition function on strings δ^* takes the values:

$$\begin{aligned} \delta^*(q_i, (\epsilon)) &= q_i \\ \delta^*(q_i, a) &= \delta(q_i, a) \quad \forall a \in \Sigma \\ \delta^*(q_i, xa) &= \delta(\delta^*(q_i, x), a) \quad \forall a \in \Sigma, x \in \Sigma^* \end{aligned}$$

That certainly was terse. But δ^* is really just what one expects it to be. It merely applies the transition function to the symbols in the string.

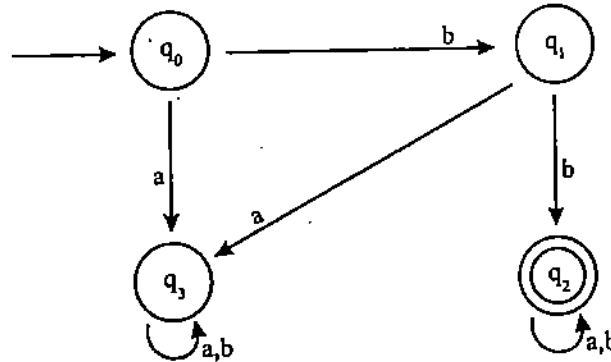


Fig. 6: Finite automata

This machine has a set of states = $\{q_0, q_1, q_2, q_3\}$ and operates over the input alphabet $\{a,b\}$. It's starting state is q_0 which can also be shown by an arrow headed toward it with no start point (as shown in Fig.6) and its set of final or accepting states, $F = \{q_2\}$ an accepting state can also be shown by two concentric circles as shown in the fig.. The transition function is fully described twice once in figure 6 as a state graph and once in table 4 as a state table.

Table 4

State	Input		Accept?
	A	b	
0	3	1	No
1	3	2	No
2	2	2	Yes
3	3	3	No

18-1

If the machine receives the input bbaa, it goes through the sequence of states:

$$q_0, q_1, q_2, q_2, q_2$$

While when it gets an input such as abab, it goes through the state transition:

$$q_0, q_3, q_3, q_3, q_3$$

Now we shall become a bit more abstract. When a finite automaton receives an input string such as:

$$x = x_1x_2\dots x_n$$

where the x_i are symbols from its input alphabet, it progresses through the sequence:

$$q_{k_1}, q_{k_2}, \dots, q_{k_{n+1}}$$

where the states in the sequence are defined as:

$$q_{k_1} = q_0$$

$$q_{k_2} = \delta(q_{k_1}, x_1) = \delta(q_0, x_1)$$

$$q_{k_3} = \delta(q_{k_2}, x_2) = \delta^*(q_0, x_1x_2)$$

$$q_{k_{n+1}} = \delta(q_{k_n}, x_n) = \delta^*(q_0, x_1x_2\dots x_n)$$

Getting back to a more intuitive reality, the following table provides an assignment of values to the symbols used above for an input of bbaba to the finite automaton of figure 3.

i	1	2	3	4	5	6
---	---	---	---	---	---	---

x_i	b	b	a	b	a	
q_{k_1}	q_0	q_1	q_2	q_2	q_2	q_2

Definition: The set (of strings) accepted by the finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is $T(M) = \{x / \delta^*(q_0, x) \in F\}$

This set of accepted strings ($L(M)$ to mean for language accepted by M) is merely all of the strings for which M ends up in a final or accepting state after processing the string. For our example (figure 3) this was all strings of 0's and 1's that contain exactly one 1. Our example (figure 6) accepted the set of strings over the alphabet $\{a,b\}$ which began with exactly two b's.

1.4.3 Finite Automata as Output Devices

The automata that we have discussed so far have only a limited output capability to the extent that only outputs are 'accepted' and 'not accepted' to indicating the acceptance or rejection of an input string. We want to introduce two classic models for finite automata that have additional output capability. We will consider machines that transform input strings into output strings. These machines are basically DFAs, except that we associate an output symbol with each state or with each state transition. But there are no final states because we are not interested in acceptance or rejection.

Mealy and Moore Machines

The first model invented by Mealy [1955] is called a Mealy machine. It associates an output letter with each transition. For example, if the output associated with the edge labelled with the letter a is x , we shall write a/x on that edge. A state transition for a Mealy machine can be presented in figure 7 as follows:

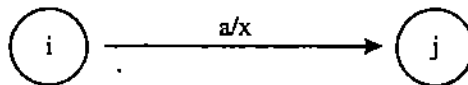


Fig. 7: Mealy machine

Indicating that the machine in state i and on input a gives output x and enters state j .

In a Mealy machine, an output always takes place during a transition of the states. The second model invented by Moore [1956], is called a Moore machine. It associates an output letter with each state. For example, if the output associated with state i is x , we will always write i/x inside the state circle. A typical state transition for a Moore machine can be presented in figure 8 as follows:

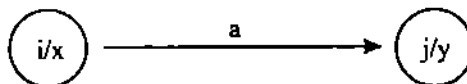


Fig. 8: Moore machine

In a Moore machine, each time a state is entered, simultaneously an output takes place. So, the first output always occurs as soon as the machine is started. Mealy and Moore machines are equivalent. In other words, any problem that is soluble by one type of machine can also be solved by the other type of machine.

Example 18: Suppose we want to compute the number of sub strings of the form bab

that occurs in an arbitrary input string over the alphabet $\{a,b\}$. For example, there are three such sub strings b, a, b in the string bab .

The diagrammatic representation of a Mealy machine for the task is given below in figure 9:

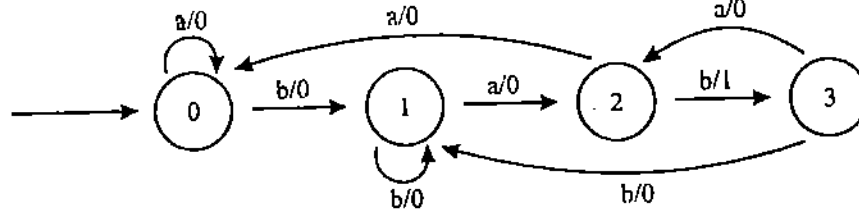


Fig. 9: Mealy machine

For example, the output of this Mealy machine for the sample string Abababaababb is 000101000010, where each 1 indicates the availability of a (or an additional) substring up to that point. On the other hand, a 0 indicates that the three previous inputs including the current input do not form a substring of the form bab.

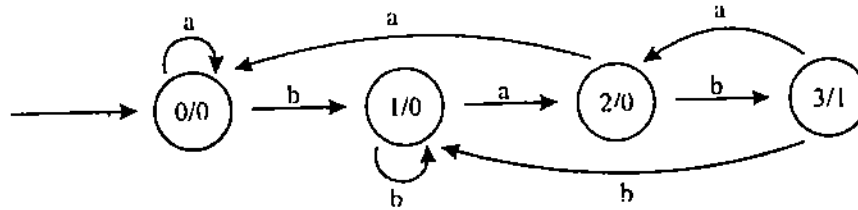


Fig. 10: Moore machine

For example, the output of this Moore machine for the simple string, Abababaababb is 0000101000010. We can count the number of 1's in the output string to obtain the number of occurrence of the sub string bab.

Example 19: A Simple Traffic Signal : Suppose we have a simple traffic intersection, where a north-south highway intersects an east-west highway. We will assume that the east-west highway always has a green light unless some north-south traffic is detected by sensors. When north-south traffic is detected, after a certain time delay the signals change and stay that way for a fixed period of time. We are required to design an appropriate circuit to capture the desired result stated above. We construct a Moore machine as a model of the required circuit as follows:

The input symbols for the required Moore machine are 0 (no traffic detected) and 1 (traffic detected). Let G, Y and R mean the colours Green, Yellow and Red, respectively. The output strings are GR, YR, RG, AND RY, where the first letter of a string is the colour of the east-west light and the second letter of a string is the colour of the north-sought light. The Moore machine model for this simple traffic intersection problem is given below diagrammatically:

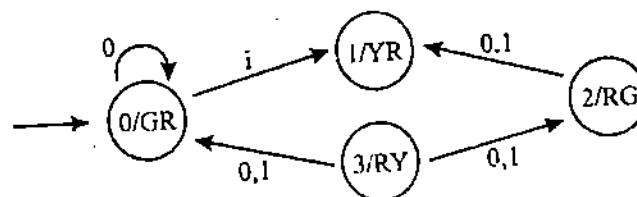


Fig. 11: Traffic signal transition diagram

Mealy machines appear to be more useful than Moore machines. But problems like traffic signal control have hic Moore machine solutions because each state is associated with a new output configuration.

Let us try some exercises:

Ex.8) Build a new FA that accepts only the word \wedge . A]so write the corresponding regular expression.

- Ex.9) Build an FA that accepts only those words that have even lengths. Also write the regular expression.
- Ex.10) Build an FA that accepts only the word haa, ab and abb and no other words. Also write the corresponding regular expression.
- Ex.11) Build an FA that will accept the language of all words each having twice as many a's as the number of b's. Also write the corresponding regular expression.
- Ex.12) Describe the languages accepted by the following FA's:

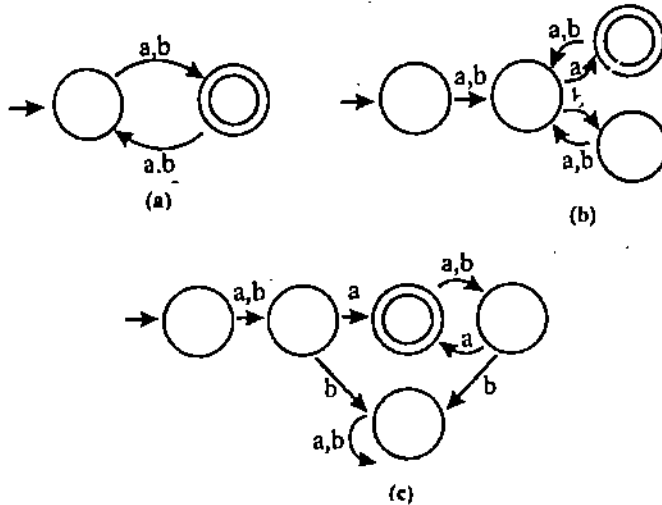


Fig. 12

1.5 SUMMARY

In this unit we introduced several formulations for regular languages, regular expressions are algebraic representations of regular languages. Finite Automata are machines that recognise regular languages. From regular expressions, we can derive regular languages. We also made some other observations. Finite automata can be used as output devices - Mealy and Moore machines.

1.6 SOLUTIONS/ANSWERS

- Ex.1) (i) ababbbaa
 (ii) baaahabb
 (iii) ab abb ab abb
 (iv) baa baa
 (v) ababbababb haa

- Ex.2) (i) Suppose $aa = x$
 Then $\{x, b\}^* = \{\Lambda, x, b, xx, bb, xb, bx, xxx, bxx, xbx, xxb, bbx, bxb, xbb, bbb\}$
 substituting $x = aa$
 $\{aa, b\}^* = \{\Lambda, aa, b, aaaa, bb, aab, bba, aaaaa, baaaa, aabaa, \dots\}$
 (ii) $\{a, ba\}^* = \{\Lambda, a, ba, aa, baba, aba, baa, \dots\}$

- Ex.3) (a) a^+b^+c
 (b) ab^+ba^+

(c) $\wedge + a(bb)^*$

Ex.4) $0+1(0+1)^*$

Ex.5) Starting with the left side and using properties of regular expressions, we get

$$\begin{aligned} & b^*(abb^* + aabb^* + aaabb^*) \\ &= b^*((ab+aab+aaab)b^*) \text{ (property 9)} \\ &= (b + ab + aab + aaab)^* \text{ (property 7)}. \end{aligned}$$

- Ex.6) (a) $\{a,b\}$
 (b) $\{a,\wedge,b,bb,\dots,b^n,\dots\}$
 (c) $\{a,b,ab,bc,abb,bcc,\dots,ab^n,bc^n,\dots\}$

- Ex.7) (a) $(aa+ab+ba+bb)^*$
 (b) $(a+b)^*aba(a+b)^*$

Ex.8)

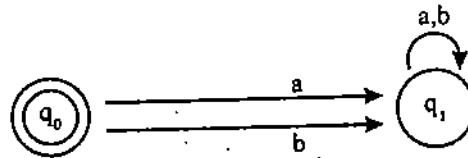


Fig. 13: Regular Expression of a null string \wedge .

Ex.9)

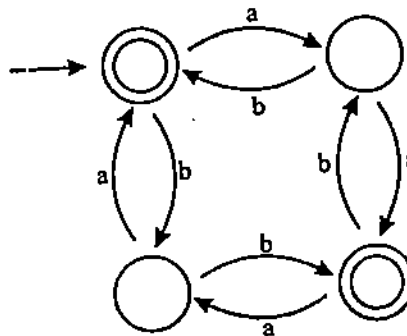


Fig. 14: Regular Expression is $(aa+ba+ab+bb)^*$

Ex.10) R.E. is $(baa + ab + abb)$

- Ex.11) (i) All the words of odd lengths.
 (ii) All the words ended with a.
 (iii) All the words with a at even places.

UNIT 2 NON-DETERMINISTIC FINITE AUTOMATA

Structure	Page Nos.
2.0 Introduction	25
2.1 Objectives	25
2.2 Non-Deterministic Finite Automata NFA	26
2.3 Equivalence of NFA and DFA	29
2.4 Equivalence of Λ -NFA's and NFA's	34
2.5 Pumping Lemma	34
2.6 Closure Properties (Regular Languages and Finite Automata)	37
2.7 Equivalence of Regular Expression and FA	42
2.8 Summary	48
2.9 Solutions/Answers	48

2.0 INTRODUCTION

In our daily activities, we all encounter the use of various sequential circuits. The elevator control which remembers to let us out before it picks up people going in the opposite direction, the traffic-light systems on our roads, trains and subways, all these are examples of sequential circuits in action. Such systems can be mathematically represented by Finite state machines, also called finite automata or other powerful machine like turning machines. In the previous unit, we introduced the concept of Deterministic Finite Automata (DFA), in which on an input in a given state of the DFA, there is a unique next state of DFA. However, if we relax the condition of uniqueness of the next state in a finite automata, then we get Non-Deterministic Finite Automata (NFA).

A natural question which now arises is whether a non-deterministic automata can recognize sets of strings which cannot be recognized by a deterministic finite automata. At first, you may suspect that the added flexibility of non-deterministic finite automata increases their computational capabilities. However, as we shall now show, there exists an effective procedure for converting a non-deterministic FA into an equivalent deterministic one. This leads us to the conclusion that non-deterministic FA's and DFA's have identical computational capabilities.

2.1 OBJECTIVES

After studying this unit, you should be able to

- define a non-deterministic finite automata;
- show the equivalence of NFA and DFA;
- compute any string or language in any NFA;
- state and prove pumping lemma;
- apply pumping lemma for a language which is not regular;
- apply closure properties of regular language and finite automata; and
- find an equivalent regular expression from a transition system and vice-versa.

In unit 1 we discussed about finite automata. You may wonder that in finite automata for each input symbol there exists a unique state for processing of it. Do you think that there may be more than one possible state, or there may not be any state for

processing of any letter. If for processing of any letter there is more than one state or none state, then, the automata is known as non-deterministic finite automata (NFA).

2.2 NON-DETERMINISTIC FINITE AUTOMATA

You have already studied finite automata (though 'automata' is a plural form of the noun 'automaton', the word 'automata' is also used in singular sense). Now consider an automata that accepts all and only strings ending in 01, represented diagrammatically, as follows:

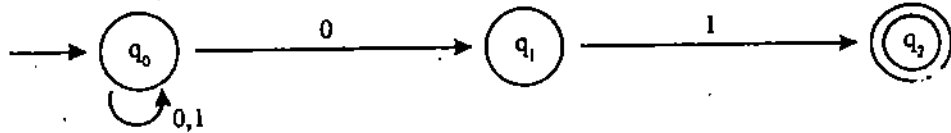


Fig. 1: Transition Diagram

In the case of the finite automata shown in figure 1, the following points may be noted:

- (i) On input 0 in state q_0 , the next state may be either of the two states viz., q_0 or q_1 .
- (ii) There is no next state on input 0 in the state q_1 .
- (iii) There is no next state on input 0 and 1 in the state q_2 .

In this transition system, what happens when this automata processes the input .00101?

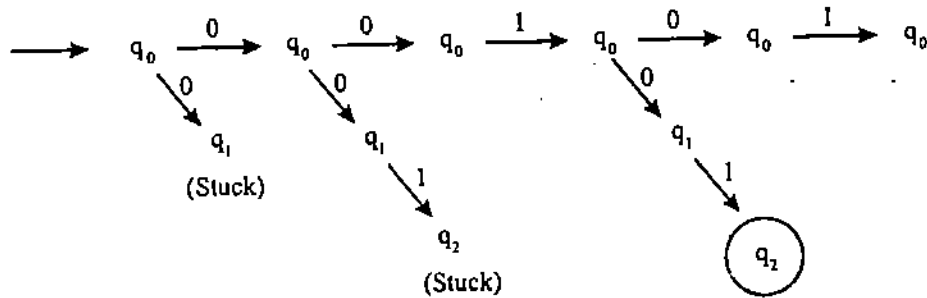


Fig. 2: Processing of string 00101

Here from the initial state q_0 , for the processing of alphabet 0, there are two states at once or viewed another way, it can be 'guessed' which state to go to next. Such a finite automata allows to have a choice of 0 or more next states for each state input pair and is called a non-deterministic finite automata. An NFA can be in several states at once.

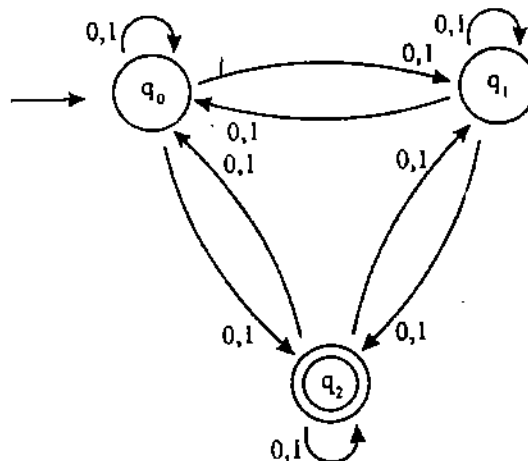
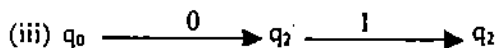
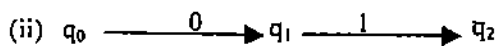
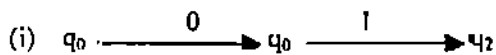


Fig. 3: Transition diagram.

Before going to the formal definition of NFA, let us discuss one more case of non-determinism of finite automata. Suppose $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, q_0 is an initial state and q_2 is final state. Again, suppose the processing of any input symbol does not result in the transition to a unique state, but results a chain of states. Let us consider a machine given in figure 3.

For the sake of convenience, let us check the processing of any input symbol. From the state q_0 , after processing 0, resulting states are q_0, q_1, q_2 and for input symbol 1, there are three possible states q_0, q_1 and q_2 not a unique state. It clarifies that a non-deterministic automata can have more than one possible state or none state after processing any input symbol from Σ .

Let us check how the string 01 is processed by the above automata. Here we have three paths to reach to the final state:



A generalisation which is obtained here by allowing of several states as a result of the processing of an input symbol is called non-determinism. If from any state, we can reach to several states or none state, then the finite automata becomes non-deterministic in nature.

Formally, a non-deterministic finite automata is a quintuple

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

Where

- * Q is a finite set of states
- * Σ is a finite alphabet for inputs
- * δ is a transition function from $Q \times \Sigma$ to the power set of Q i.e. to 2^Q
- * $q_0 \in Q$ is the start/initial state
- * $F \subseteq Q$ is a set of final/accepting states.

The NFA, for the example just considered, can be formally represented as:

$$(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$

where δ the transition function, is given by the table 1:

Table 1

States	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$
q_2	\emptyset	\emptyset

Now, let us prove that the NFA

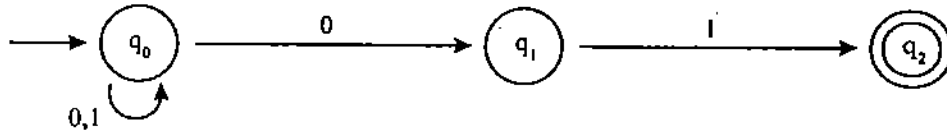


Fig. 4: NFA accepting x01

accepts the language $\{x01 : x \in \Sigma^*\}$ of all the strings that terminate with the sub-string 01. A mutual induction on the three statements below proves that the NFA accepts the given language.

1. $w \in \Sigma^* \Rightarrow q_0 \in \delta(q_0, w)$
2. $q_1 \in \delta(q_0, w) \Leftrightarrow w = x0$
3. $q_2 \in \delta(q_0, w) \Leftrightarrow w = x01$

If $|w| = 0$ then $w = \Lambda$. Then statement (1) follows from def., and statement & (2) and (3) show that all the string x01 will be accepted by the above non-deterministic automata.

Example 1: Consider the NFA with the formal description as $(Q, \Sigma, \delta, q_0, F)$ where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, q_0 is the initial state and q_1 is only the final state, and δ is given by the following table:

Table 2

State	Input from Σ	
	a	b
$\rightarrow q_0$	q_1, q_2	q_0
q_1	q_1	q_1
q_2	q_1	q_2

In NFA, though the function δ maps to a sub-set of the set of states, yet we generally drop braces, i.e., instead of $\{q_0, q_1\}$, we just write q_0, q_1 .

The computation for an NFA is also similar to that of DFA. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and w is a string over the alphabet Σ . The string w is accepted by NFA if corresponding to the input sequence, there exists a sequence of transitions from the initial state to any of the possible final states.

\therefore Now, let us check computations (in NFA, there are many possible computations) of the string aba.

$$\begin{aligned}
 \delta(q_0, aba) &= \delta(\delta(q_0, a), ba) \\
 &= \delta(q_1, ba) \text{ or } \delta(q_2, ba) \\
 &= \delta(\delta(q_1, b), a) \text{ or } \delta(\delta(q_2, b), a) \\
 &= \text{stuck or } \delta(q_2, a) \\
 &= q_1 \text{ (an accepting state)}
 \end{aligned}$$

The above sequence of states shows the final state q_1 which is an accepting state. Hence, the string aba is accepted by the system and the input sequence of states for the

$$\text{input is } \rightarrow q_0 \xrightarrow{a} q_2 \xrightarrow{b} q_2 \xrightarrow{a} q_1$$

Try some exercises:

Ex.1) Consider an NFA given in figure 5. Check whether the strings 001, 011101, 01110, 010 are accepted by the machine, or not?

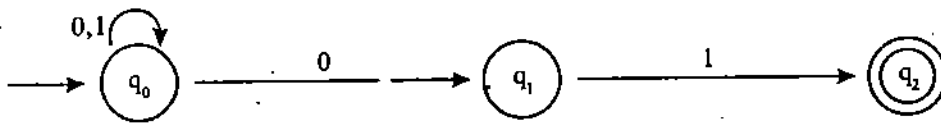


Fig. 5

Ex.2) Give an NFA which accepts all the strings starting with ab over {a,b}.

In unit 1, we discussed DFA and in previous section we discussed NFA. Now a simple question arises, are these two automata equivalent? Reply for that is it is always possible to find an equivalent DFA to every NFA. In next section we shall discuss the equivalence of DFA and NFA.

2.3 EQUIVALENCE OF NFA AND DFA

Every time we find that if we are constructing an automata, then it is quite easy to form an NFA instead of DFA. So, it is necessary to convert an NFA into a DFA and this is also said to be equivalence of two automata. Two finite automata M and N are said to be equivalent if $L(M) = L(N)$.

From the definitions of NFA and DFA, it is clear that they are similar in all respects except for the transition function. In DFA, the transition function takes a state and an input symbol to the next state, whereas in NFA, the transition function takes a state and an input symbol or the empty string into the set of possible next states. If empty string Λ is used as an input symbol, then the NFA is called Λ -NFA. As an NFA is obtained by relaxing some condition of DFA, intuitively it seems that there may be some NFAs to which no DFA may correspond. However, it will be shown below that by relaxing the condition, we are not able to enhance computational power of the DFAs. In other words, we establish that for each NFA, there is a DFA, so that both recognise/accept the same set of strings.

We now try to find the equivalence between DFA and NFA. Some DFA can be designed to simulate the behaviour of an NFA. Let us consider $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA accepting $L(M)$. We design a DFA, viz., M' as described below and show that the language accepted by M' is the same that accepted by M, i.e., the language $L(M)$. $M' = (Q', \Sigma, \delta', q'_0, F')$ where $Q' = 2^Q$ (any state in Q' is denoted by $[q_1, q_2, \dots, q_j]$ where $q_1, q_2, \dots, q_j \in Q$), $q'_0 = [q_0]$ and F' is the set of all subsets of Q containing an element of F.

Before defining δ' , let us look at the construction of Q' , q'_0 and F' . Machine M is initially at q_0 state. But on application of an input symbol, say a, M can reach any of the states in $\delta(q_0, a)$. So M' has to remember all these possible states at any point of time. Therefore, subsets of Q can be defined as the states of M' . Initial state of M' is q_0 , which is defined as $[q_0]$. A string w accepted by the machine M if a final state is one of the possible states M reaches on processing w. So, a final state in M' is any subset of Q containing some final state of M. Next we can define the transition function δ' as

$$\delta'([q_1, \dots, q_N], a) = \bigcup_{i=1}^N \delta(q_i, a).$$

So, we have to apply δ to (q_i, a) for each $i = 1, 2, \dots, N$

and take their union to get $\delta'([q_1, q_2, \dots, q_N], a)$. Defining δ' with the help of δ in this way is also said to be subset construction approach.

Example 2: Construct a DFA equivalent to the NFA M , diagrammatically given by

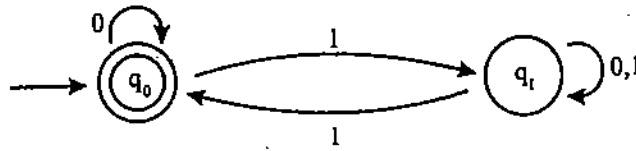


Fig. 6: NFA

when δ for M is given in terms of a transition table, the construction is simpler. Now, let us have a look at the following table 3.

Table 3

State/ Σ	0	1
$\rightarrow q_0$	q_0	q_1
q_1	q_1	q_0, q_1

- (i) In this given M , the set of states is $\{q_0, q_1\}$. The states in the equivalent DFA are the subsets of the states given in the NFA. So the states in DFA are subsets of $\{q_0, q_1\}$, i.e., $\phi, [q_0], [q_1], [q_0, q_1]$.
- (ii) $[q_0]$ is the initial state.
- (iii) $[q_0]$ and $[q_0, q_1]$ are the final states as these are the only states containing q_0 , the only final state of M .

Therefore, $F' = \{[q_0], [q_0, q_1]\}$

(iv) δ' is defined by the following state table:

Table 4

State/ Σ	0	1
ϕ	ϕ	ϕ
$\rightarrow [q_0]$	$[q_0]$	$[q_1]$
$[q_1]$	$[q_1]$	$[q_0, q_1]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1]$

We start the construction by considering $[q_0]$ first. We get $[q_0]$ and $[q_1]$. Then, we construct δ for $[q_1]$ we get $[q_1]$ and $[q_0, q_1]$. As $[q_1]$ already exists in left most column, so we construct δ for $[q_0, q_1]$. We get $[q_0, q_1]$ and $[q_0, q_1]$. We do not get $[q_0, q_1]$ and $[q_0, q_1]$. We do not get any new states and so we terminate the construction of δ .

When a non-deterministic finite automata has n states, the corresponding finite automata has 2^n states. However, it is not necessary to construct δ for all these 2^n states, but only for those states reachable from the initial state. This is because our interest is only in constructing the equivalent DFA. Therefore, we start the construction of δ for initial state and continue by considering only states appearing earlier under input columns and constructing δ for such states. If no more new states appear under the input columns, we halt.

To prove the equivalence of both automata, we will prove the following theorem:

Theorem 1: A language L is accepted by some NFA if and only if it is accepted by some DFA.

In the theorem, there are two parts to prove:

If L is accepted by DFA M' , then L is accepted by some NFA M .
 If L is accepted by NFA M' , then L is accepted by some DFA M .

The first is the easier to prove.

Theorem 1(a) (one direction): If L is accepted by DFA M' , then L is accepted by some NFA M .

Proof: Let us compare the definitions of NFA and DFA.

Definition: A Deterministic Finite Automata (DFA) M' is defined by the 5-tuple.

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

Q' - The finite set of states.

Σ - The finite set of symbols, the input alphabet.

δ' - Transition function $\delta': Q' \times \Sigma \rightarrow Q'$.

q_0 - An initial state, $q_0 \in Q'$.

F' - A set of final states or accept states, $F' \subseteq Q'$.

Definition: A Non-deterministic Finite Automata (NFA) M is a 5-tuple

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

Q - is a finite set of states.

Σ - is a finite input alphabet.

δ - is a transition function $\delta: Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$.

$q_0 \in Q$ is the start state.

$F \subseteq Q$, is the set of accepting states.

The above definitions follow that every DFA is also an NFA, which implies that if $w \in L(M')$, then $w \in L(M)$.

The other half of the theorem is in the following theorem:

Theorem 1(b): If L is accepted by NFA $M = (Q, \Sigma, \delta, q_0, F)$, then L is accepted by some DFA $M' = (Q', \Sigma, \delta', q_0, F')$.

Proof: Construct M' as in the Subset Construction Algorithm. We will show using induction on the length of w .

Base case: Let w be an empty string, i.e., if $|w| = 0$ then $w = \lambda$. By definition of NFA and DFA both $\delta(q_0, w)$ and $\delta'(q_0, w)$ are in state $\{q_0\}$. Hence, the result.

Let us assume that this result is true for each string of length n , we will now show that this result is true for strings of length $(n+1)$.

Let $w = sa$ with $|w| = (n + 1)$ and $|s| = n$, also a is the final symbol for w . As $|s| = n$, therefore, by induction

$$\delta'(q_0, s) = \delta(q_0, s)$$

If $\{P_1, P_2, \dots, P_k\}$ be the set of states for non-deterministic finite automata M , then

$$\delta'(q_0, w) = \bigcup_{i=1}^k \delta(P_i, a) \tag{i}$$

Next,

$$\delta'(\{P_1, P_2, \dots, P_k\}, a) = \bigcup_{i=1}^k \delta(P_i, a) \tag{ii}$$

$$\text{also } \delta'(q_0, s) = \{P_1, P_2, \dots, P_k\} \tag{iii}$$

Using Equations (i), (ii) and (iii) we get

$$\begin{aligned} \delta'(q_0, w) &= \delta'(\delta'(q_0, s), a) \\ &= \delta'(\{P_1, P_2, \dots, P_k\}, a) \\ &= \bigcup_{i=1}^k \delta(P_i, a) \\ &= \delta(q_0, w). \end{aligned}$$

which shows that the result is true for $|w| = n + 1$ when the result is true for a string of length n .

Here the result is true for length 0 and for length $(n + 1)$ which is implied by the length n .

Therefore, the given statement is true for all the strings.

Hence, M and M' both accept the same string w iff $\delta'(q_0, w)$ or $\delta(q_0, w)$ contains a state in F' , or F respectively. Therefore,

$$L(M) = L(M')$$

For every non-deterministic finite automaton, there exists an equivalent deterministic infinite automaton which accepts the same language. In this way, two finite automata M and M' are said to be equivalent if $L(M) = L(M')$.

Example 3: Construct a non-deterministic finite automata accepting the set of all strings over $\{a, b\}$ ending in aba . Use it to construct a DFA accepting the same set of strings.

Solution: Required NFA is the one that accepts strings of the form $xaba$ where $x \in \{a, b\}^*$

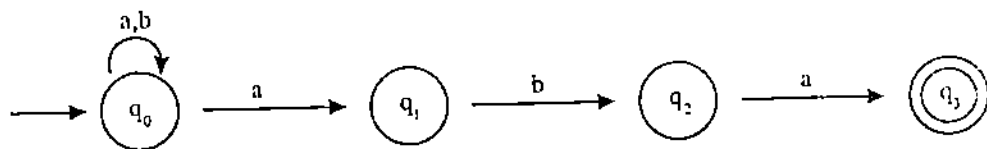


Fig. 7: NFA accepting all the string ended by aba

Transition table of the diagram shown in Figure 7 is given in table 5.

Table 5

State/ Σ	A	B
$\rightarrow q_0$	q_0, q_1	q_0
q_1	-	q_2
q_2	q_1	-
q_3	-	-

Now, let us construct its equivalent DFA. $[q_0]$ is the initial state in corresponding DFA so starting the δ function using $[q_0]$ as an initial state, we represent it in table 6.

Formally, the DFA is

$$A = (\{[q_0], [q_0, q_1], [q_0, q_2], [q_0, q_1, q_3]\}, \{a, b\}, \delta, [q_0], \{[q_0, q_1, q_3]\})$$

where δ is given by the table 6.

Table 6

State/ Σ	a	b
$\rightarrow [q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1]$
$[q_0, q_2]$	$[q_0, q_1, q_3]$	$[q_0]$
$[q_0, q_1, q_3]$	$[q_0, q_1, q_3]$	$[q_0, q_2]$

Diagrammatically, DFA is given in figure 8.

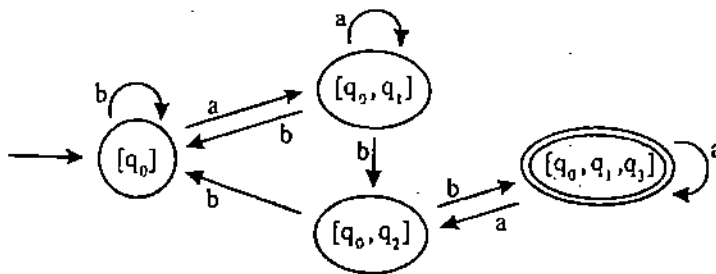


Fig. 8: DFA

This example also highlights one of the reasons for studying NFAs. The reason is that generally, it is easier to construct an NFA that accepts a language than to construct the corresponding DFA.

Try some exercises to check your understanding:

Ex.3) Construct an NFA accepting $\{01, 10\}$ and use it to find a DFA accepting the same.

Ex.4) $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \delta, q_1, \{q_3\})$ is a NFA, where δ is given by

$$\delta(q_1, 0) = \{q_2, q_3\}, \delta(q_1, 1) = \{q_1\}$$

$$\delta(q_2, 0) = \{q_1, q_2\}, \delta(q_2, 1) = \phi$$

$$\delta(q_3, 0) = \{q_2\}, \delta(q_3, 1) = \{q_1, q_2\}$$

construct an equivalent DFA

- Ex.5) Construct a transition system which can accept strings over the alphabet a, b, \dots containing either cat or rat .
- Ex.6) Give examples of machines distinguishing DFA and NFA.

2.4 EQUIVALENCE OF Λ -NFA AND NFA

There exist some transition graphs when no input is applied. If no input is applied then the transition systems are associated with a null symbol Λ . Every time we can find an equivalence in between the systems with Λ -move and without Λ -moves. With the help of an example, we shall find the equivalence of Λ -NFA and NFA.

Suppose we want to remove Λ -move from the transition shown in figure 9:

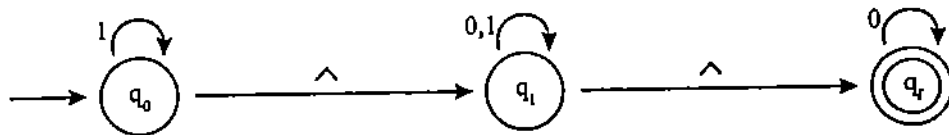


Fig. 9: Λ -NFA

In the above transition q_0 is an initial state and q_f is a final state. For this, we proceed as follows:

If q_i, q_j are two states and null string is from q_i to q_j then :

- Duplicate all the edges starting from q_i which are starting from q_j
- If q_j is a final state, make q_i as a final state and if q_i is an initial state, make q_j as an initial state.

Now let us apply these two rules to the transition in figure 9. First of all, removing Λ in between q_1 and q_f .

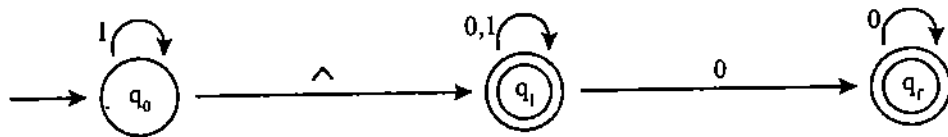


Fig. 10: Removal of one Λ

Now, again apply the same rule to remove the remaining Λ -move.

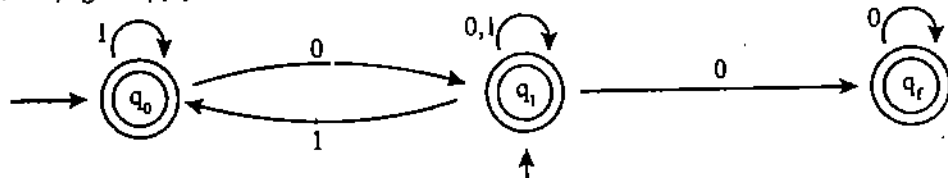


Fig. 11: After removing both Λ

This transition system is free from Λ and is equivalent to the Λ -NFA.

2.5 PUMPING LEMMA

As you know that a language which can be defined by a regular expression is called a regular language, there are several questions related to regular languages that one can ask. The important one is: are all languages regular? The simple answer is no. The languages which are not regular are called non-regular languages. In this section, we give a basic result called "pumping lemma". Pumping lemma gives a necessary condition for an input string to belong to a regular set, and also states a method of

pumping (generating) many input strings from a given strings all of which should be in the language if the language is regular. As this pumping lemma gives a necessary (but not sufficient) condition for a language to be regular, we cannot use this lemma to establish that a given language is regular, but we can use it to prove that a language is not regular by showing that the language does not obey the lemma.

The pumping lemma uses the **pigeonhole principle** which states that if p pigeons are placed into less than p holes, some hole has to have more than one pigeon in it. The same thing happens in the proof of pumping lemma. The pumping lemma is based on this fact that in a transition diagram with n states, any string of length greater than or equal to n must repeat some state.

Pumping lemma (PL):

If L is a regular language, then there exists a constant n such that every string w in L , of length n or more, can be written as $w = xyz$, where

- (i) $|y| > 0$
- (ii) $|xy| \leq n$
- (iii) $xy^i z$ is in L , for all $i \geq 0$ here y^i denotes y repeated i times and $y^0 = \Lambda$

Before proving this PL, a question that may have occurred by now is: Are there any languages that are not accepted by DFA's?

Consider the language $L = \{w \mid w=0^k 1^k, \text{ where } k \text{ is a positive integer}\}$.

Proof of (PL): Since we have L is regular, there must be a DFA, say A such that

$$L = L(A)$$

Let A have n states, and a string w of length $\geq n$ in L which is expressed as

$$w = a_1 a_2 \dots a_k \text{ where } k \geq n \text{ with general elements } a_i, a_j, \text{ for}$$

$1 \leq i \leq j \leq k$, the string w can be written as

$$w = a_1 a_2 \dots a_{i-1} a_i a_{i+1} a_{i+2} \dots a_{j-1} a_j a_{j+1} \dots a_k$$

and $w = xyz$

$$\text{so } x = a_1 a_2 \dots a_i$$

$$y = a_{i+1} a_{i+2} \dots a_j$$

$$\text{and } z = a_{j+1} a_{j+2} \dots a_k$$

Let q_0 be the initial state and further let

$$q_1 = \delta(q_0, a_1), q_2 = \delta(q_0, a_1 a_2)$$

q_i be the state in which A is after reading the first i symbols of w .

Since there are only n different states at least two of q_0, q_1, \dots, q_n which are $(n+1)$ in numbers, must be same say, $q_i = q_j$ where $0 \leq i < j \leq n$. Then by repeating the loop from q_i to q_j with label $a_{i+1} \dots a_j$ zero times once, or more, we get $xy^i z$ is accepted by A , because in case of each of the string $xy^i z$ for $i = 1, 2, \dots$, the string when given as an

input to the machine in the initial state q_0 , reaches the final state q_n .
Diagrammatically.

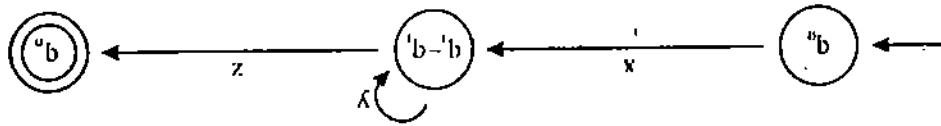


Fig. 12: representation of $xy^i z$

Hence $xy^i z \in L(A) \forall i \geq 0$.

How to use PL in establishing a given language as non-regular?

We use the PL to show that a language L is not regular through the following sequence of steps:

- Step1: Start by assuming L is regular.
- Step2: Suppose corresponding DFA has n states.
- Step3: Choose a suitable w such that $w \in L$ with $|w| \geq n$.
- Step4: Apply PL to show that there exists $i \geq 0$ such that $xy^i z \notin L$, where $w = xyz$ for some strings xyz .
- Step5: Thus, we derive a contradiction by picking i , which concludes that assumption in step 1 is false.

Example 4: Consider $L = \{0^{n^2} | n \geq 0\}$.

Suppose L is regular. Then there exists a constant n satisfying the PL conditions.

Now $w = 0^{n^2} \in L$ and $|w| = n^2$

Write $w = xyz$; where $|xy| \leq n$ and $|y| > 0$ and hence $|y| \leq n$

By PL, $xyyz \in L$.

Here $|w| = n^2$

$$\Rightarrow |xyz| = n^2$$

$$\Rightarrow |x| + |y| + |z| = n^2$$

$$\Rightarrow n^2 + n \geq |x| + |y| + |y| + |z| > n^2; \text{ [as } |x| > 0 \text{ and } |xy| \leq n]$$

$$\Rightarrow n^2 + n \geq |xyyz| > n^2$$

$$\Rightarrow (n+1)^2 > |xyyz| > n^2.$$

$(n+1)^2$ is the next perfect square after n^2 , therefore,

$xyyz$ is not of square length and is not in L . Since we have derived a contradiction, which concludes that L is not regular.

Let us try some exercises:

Ex.7) Show that the following languages are not regular

- (i) $L_1 = \{0^m 1^m : m \geq 0\}$
- (ii) $L_2 = \{0^i 1^j 2^k : 0 \leq i, j < k\}$
- (iii) $L_3 = \{a^p : p \text{ is prime}\}$
- (iv) $L_4 = \{ww \mid w \in \{0,1\}^*\}$
- (v) $L_5 = \{0^n 1^{n!} \mid n > 0\}$

Ex.8) Give an example of a language which is not regular. Justify your answer.

2.6 CLOSURE PROPERTIES (Regular Languages and Finite Automata)

Suppose L and M are two regular languages, then if the operations applied to L and M results regular language, then the property is called closure property. The closure properties are very useful for regular languages and finite automata. The operations applied for regular languages produce regular language are union, intersection, concatenation, complementation, Kleenstar and difference. With the help of closure properties, we can easily construct the finite automata which accepts the language which is union, intersection, ..., of regular languages.

Before discussing the closure properties, let us define a language of a DFA. Suppose $M = (Q, \Sigma, \delta, q_0, F)$, and the language accepted by M is $L(M)$ and is defined as $L(M) = \{S \mid \delta^*(q_0, S) \in F\}$. That is each string in $L(M)$ is accepted by M. If $L = L(M)$, then L is regular language. Let us discuss few theorems, showing the closure properties of regular languages and finite automata.

Theorem2: If L and M are regular languages, then $L+M$, LM and L^* are also regular languages.

L and M being given to be regular languages can be denoted by some regular expressions, say, l and m . Then, $(l+m)$ denotes the language $L+M$. Also, the regular expression lm denotes the language LM . $(l)^*$ denotes the language L^* . Therefore, all three of these sets (i.e., languages) of words are definable by regular expressions, and hence are themselves regular languages.

Note: If any language can be denoted by a regular expression, then that language is by definition a regular language.

Complements and Intersection

Definition: If L is a language over the alphabet Σ , we define its complement, \bar{L} , to be the language of all strings of letters from Σ^* that are not in L, i.e., $\bar{L} = \Sigma^* - L$.

Example 5: Let L be the language over the alphabet $\Sigma = \{a,b\}$ having all the words which start with the letter a and no other words over Σ . Then, \bar{L} is the language of the all other words that do not have the first letter as a.

Example 6: Suppose L is a language over $\{a,b\}$ ending with ba , then \bar{L} is the language of over $\{a,b\}$ of all other words not ending with ba .

Theorem 3: If L is a regular language, then \bar{L} is also a regular languages. In other words, the set of regular languages is closed under complementation.

Proof : We establish the result by constructing an FA say \bar{M} , the language \bar{L} . As L is given to be regular, therefore there is as FA, say M that recognizes L .

If $L = \Sigma^*$, then $\bar{L} = \phi$, which is, by definition, a regular language.

If $L \neq \Sigma^*$ is a regular language, then there is some FA that accepts the language L .

At least one of the states of the FA is a final state and as $L \neq \Sigma^*$, at least one of the states must not be a final state. The required FA has the same set of states, same set of input symbols, same transition function and same initial state as M . However, if S is the set of all states of M and F is the set of all final states of M , then set $S \sim F$ of all non-final states of M serves as set of final states of the proposed FA viz. \bar{M} .

The fact that \bar{M} is the FA that recognises the language \bar{L} , follows from the following:

Let $x \in \bar{L} = \Sigma^* \sim L \Leftrightarrow x \notin L$

\Leftrightarrow the string x when given to M as input string in the initial state terminates in a non-final state of M , i.e., terminates in a state belonging to $S \sim F$.

$\Leftrightarrow \bar{M}$ accepts x

Theorem 4: If L and M are regular languages, then $L \cap M$ is also a regular language. In other words, the set of regular languages is closed under intersection.

Proof: We can prove this theorem in two ways: One by De Morgan's Law or by constructing an appropriate FA. Here the proof with the help of De Morgan's law is given, and leave the proof based on construction of an appropriate FA to the students as an exercise.

For any two general sets L and M , whether regular languages or not, by De Morgan's Laws, we have

$$L \cap M = \overline{(\bar{L} + \bar{M})}.$$

In view of the fact that complement of a regular language is regular, the languages \bar{L} and \bar{M} are regular languages, given L and M are regular. Further, the fact that the sum of two regular languages is regular, makes $\bar{L} + \bar{M}$ as a regular language.

Hence, its complement $\overline{(\bar{L} + \bar{M})} = L \cap M$, is regular.

The following discussion, based on processing of two FAS in parallel, helps us in the construction of an FA for the union of two regular languages.

Example 7: Suppose we take the two machines whose state graphs are given in the figure below:

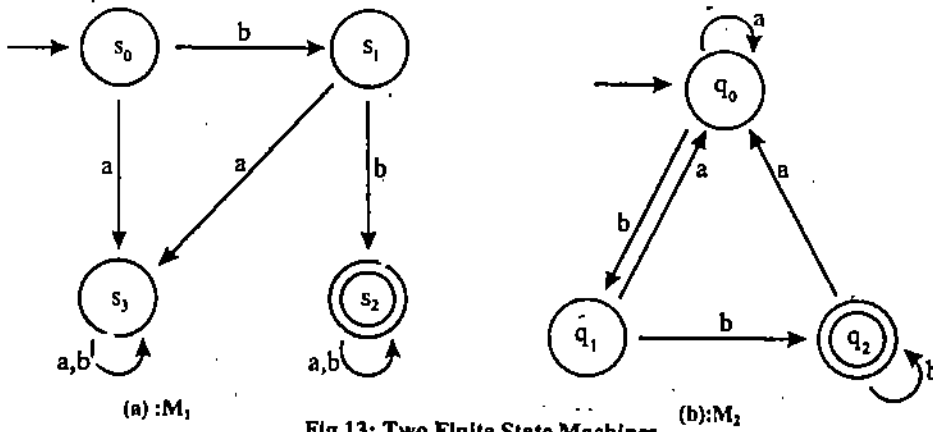


Fig.13: Two Finite State Machines

We can easily verify that the machine (M_1) of figure 14 accepts all strings (over $\{a, b\}$) which begin with two b's. The other machine (M_2) in figure 15 accepts strings which end with two b's. Let's try to combine them into one machine which accepts strings which either begin or end with two b's.

Why not run both machines at the same time on an input? We could keep track of what state each machine is in, by placing pebbles upon the current states and then advancing them according to the transition functions of each machine. Both machines begin in their starting states, as pictured in the state graphs below:

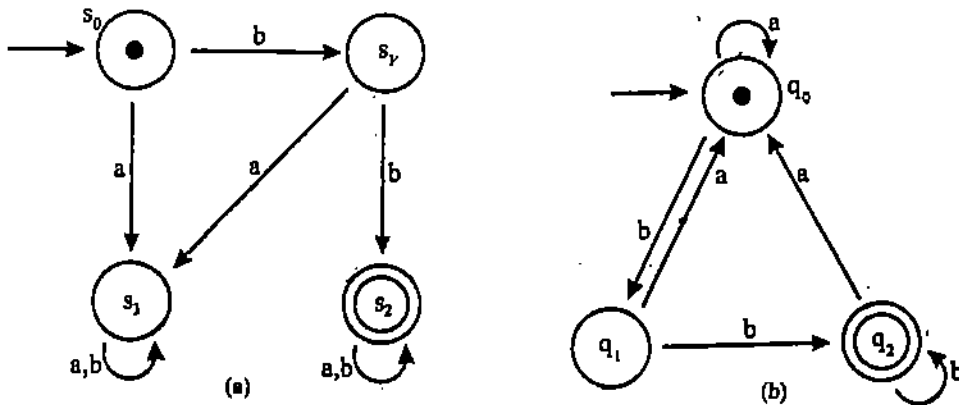


Fig. 14: Pebble on s_0 and q_0

With pebbles on s_0 and q_0 , if both machines now read the symbol b on their input tapes, they move the pebbles to new states and the machines assume the following configurations:

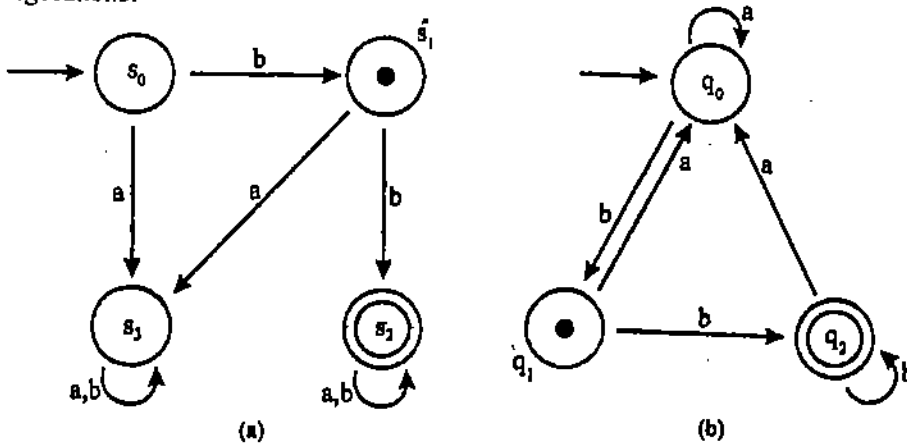


Fig. 15: Pebble on s_1 and q_1

with pebbles on s_1 and q_1 . The pebbles have advanced according to the transition functions of the machines. Now let's have them both read an a . At this point, they both advance their pebbles to the next state and enter the configurations

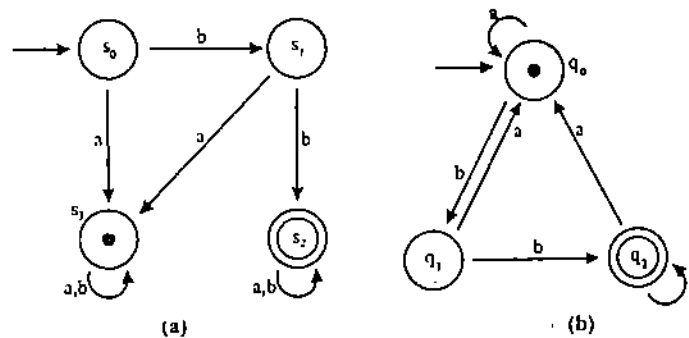


Fig. 16: Pebble on s_1 and q_0

With this picture in mind, let's trace the computations of both machines as they process several input strings. Pay particular attention to the *pairs* of states the machines go through. Let our first string be $bbabb$, which is accepted by both the machines.

Table 7

Input	B	b	a	b	b
M_1 's states	s_0	s_1	s_2	s_2	s_2
M_2 's states	q_0	q_1	q_2	q_0	q_1

Now, let us look at an input string which neither of the two machines accepts say $babab$.

Table 8

Input	b	a	b	a	b
M_1 's states	s_0	s_1	s_3	s_3	s_3
M_2 's states	q_0	q_1	q_0	q_1	q_1

And finally, we consider the string $baabb$ which will be accepted by M_2 but not M_1 .

Table 9

Input	b	a	a	b	b
M_1 's states	s_0	s_1	s_3	s_3	s_3
M_2 's states	q_0	q_1	q_0	q_0	q_1

If we imagine a multi-processing finite automaton with two processors (one for M_1 and one for M_2), it would probably look just like the pictures given above. Each of its state is a pair of states, one from each machine, corresponding to the pebble positions. Then, if a pebble ended up on an accepting state, for either machine (that is, either s_2 or q_2), our multi-processing finite automaton would accept the string. The above discussion helps us in seeing the truth of the following statement intuitively: We construct the required machine by simulating the multi-processing pebble machine discussed above.

Theorem 5: The class of sets accepted by finite automata is closed under union.

Proof Sketch : Let $M_1 = (S, \Sigma, \delta, s_0, F)$ and $M_2 = (Q, \Sigma, \gamma, q_0, G)$ be two arbitrary finite automata. To prove the theorem, we must show that there is another machine (M_3) which accepts every string accepted by M_1 or M_2 and no other string.

We show that the required machine is $M_3 = (S \times Q, \Sigma, \xi, \langle s_0, q_0 \rangle, H)$ where ξ and H will be described presently.

The transition function ξ is defined as

$$\xi (<s_i, q_i>, a) = <\delta(s_i, a), \gamma(q_i, a)>.$$

It can easily be seen that ξ is a function from $S \times Q$ to $S \times Q$.

A state in M_3 is a final state in M_3 if and only if either its first component is in F , i.e., is a final state of M_1 or its second component is in G , i.e., is a final state of M_2 . In cross product notation, this is :

$$H = (F \times Q) \cup (S \times G).$$

This completes the definition of M_3 . We can easily see that M_3 is indeed a finite automaton because it satisfies the definition of finite automata. We claim it does accept $T(M_1) \cup T(M_2)$ since it mimics the operation of our intuitive multi-processing pebble machine. The remainder of the formal proof (which we shall leave as an exercise) is merely an introduction on the length of input strings to show that for all strings x over the alphabet I :

$$\begin{aligned} x \in T(M_1) \cup T(M_2) &\text{ iff } \delta^*(s_0, x) \in F \text{ or } \gamma^*(q_0, x) \in G \\ &\text{ iff } \xi^*(<s_0, q_0>, x) \in H. \end{aligned}$$

Thus, by construction we have shown that the class of sets accepted by finite automata is closed under union.

By manipulating the notation, we have shown that two finite automata given in figure 13 (a) and (b) can be combined in a special way to prove the desired result, as shown in figure 17.

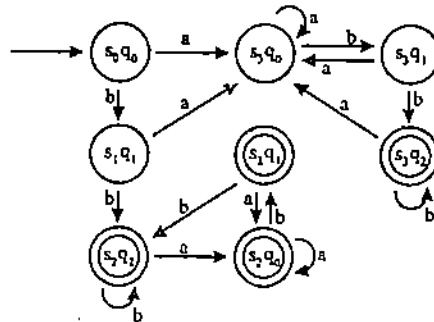


Fig. 17: Union of M_1 and M_2

Note that not all pairs of states are included in the state graph. (For example, $<s_0, q_1>$ and $<s_1, q_2>$ are missing.) This is because it is impossible to get to these states from $<s_0, q_0>$.

This is indeed a complicated machine! But, if we are a bit clever, we might notice that if the machine enters state s_2, q_2 , then it remains in one of the states $(s_2, q_0), (s_2, q_1), (s_2, q_2)$ all of which are final states. We may replace all such stages of M_3 by a single state say s_2, q_1 , which is also a final state and get a smaller but equivalent machine as shown in Figure 18:

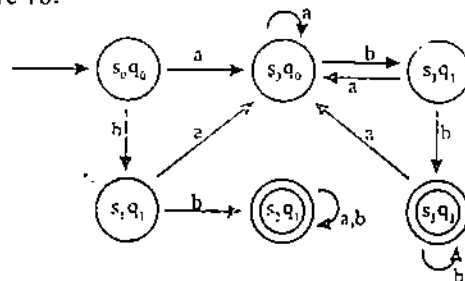


Fig. 18: Reduced Union Machine

Now check your understanding by the following exercises.

Ex. 9) For each of the following pairs of regular languages, L and M find a regular expression and an FA that correspond to $L \cap M$:

	L	M
1.	$(a+b)^*a$	$(a+b)^*b$
2.	$(a+ab)^*(a+\wedge)$	$(a+ba)^*a$
3.	$(ab)^*$	$b(a+b)^*$
4.	$(a+b)^*a$	$(a+b)^*aa(a+b)^*$
5.	All strings of even length $= (aa+ab+ba+bb)^*$	$b(a+b)^*$

2.7 EQUIVALENCE OF REGULAR EXPRESSION AND FA

As you have seen in Unit 1, all the regular languages can be written as regular expression and vice-versa. Do you find any relation in regular expression and a transition system? A regular expression can have \wedge , ϕ , any input symbol, +, *, concatenation. Let us find the transition system of these.

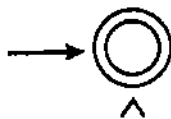


Fig. 19: Transition diagram equivalent to \wedge

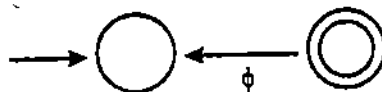


Fig. 20: Transition diagram equivalent to \emptyset

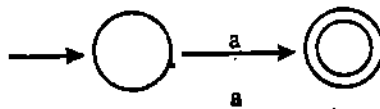


Fig. 21: Transition diagram equivalent to a

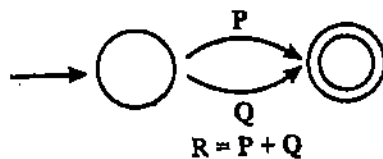


Fig. 22: Transition diagram equivalent to $R = P + Q$

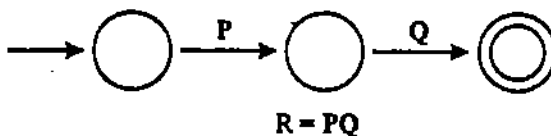


Fig. 23: Transition diagram equivalent to $R = PQ$

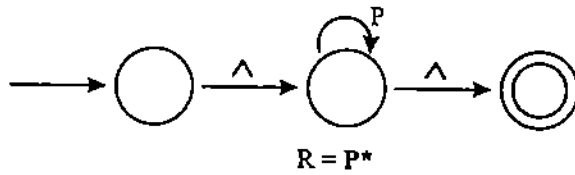


Fig. 24: Transition diagram equivalent to $R = P^*$

Using above equivalence of regular expression and transition systems, we can easily make use of equivalence of \wedge -NFA and NFA and also of NFA and DFA, and finally we can find the equivalence between a regular expression and FA.

Example 8: Let us try to get the finite automata which is equivalent to regular expression $(a+b)^*(ab+ba)(a+b)^*$.

Step 1: Construction of equivalent \wedge -NFA is:

$(a+b)^*(ab+ba)(a+b)^*$ is

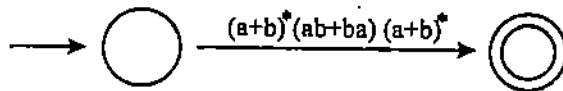


Fig. 25: A Complete regular expression

It is concatenation of $(a+b)^*$, $(ab+ba)$ and $(a+b)^*$, after applying concatenation we get

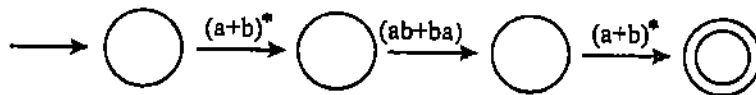


Fig. 26: After concatenation

Then removing the * from $(a+b)^*$ at both places and applying union rule for $ab + ba$ we get

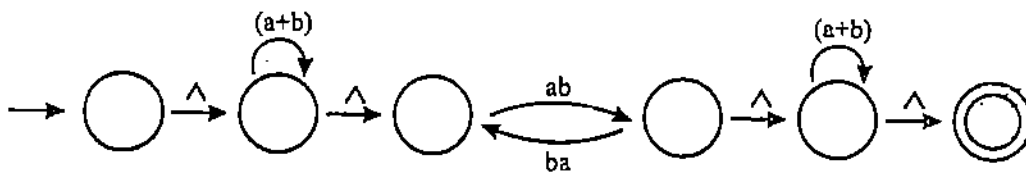


Fig. 27: after removing * and +

Now concatenating ab and ba , we get

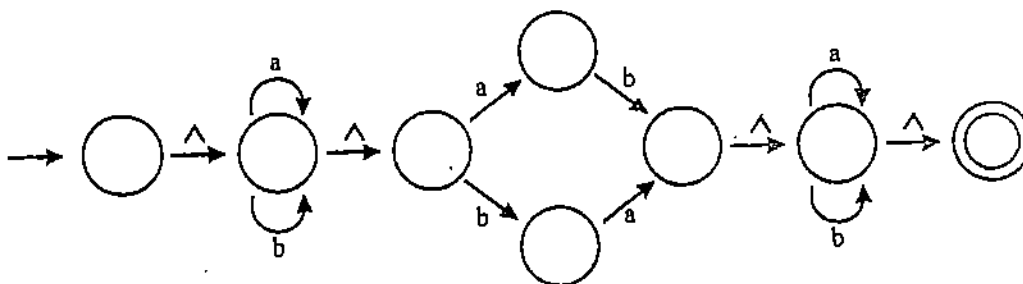


Fig. 28: equivalent \wedge -NFA

Step 2 : Construction of equivalent NFA, Let us remove every \wedge one by one

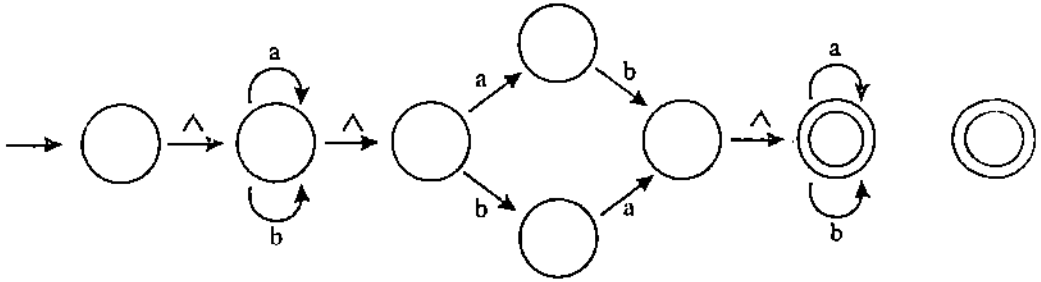


Fig. 29: Removing \wedge

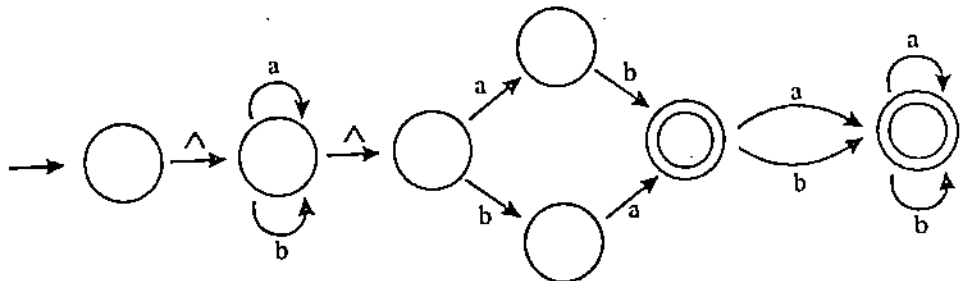


Fig. 30: Removing \wedge and Minimizing the state

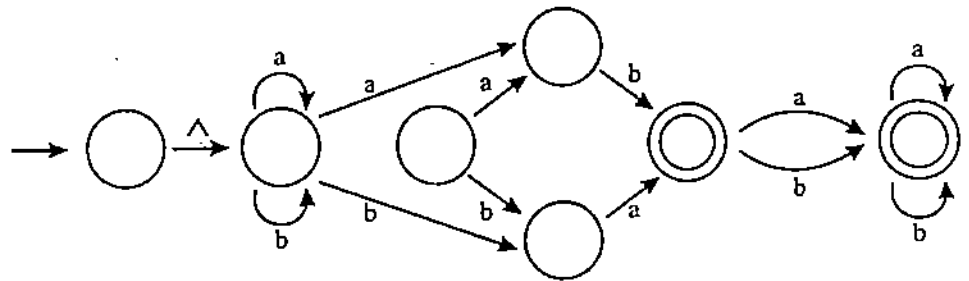


Fig. 31: Removing \wedge

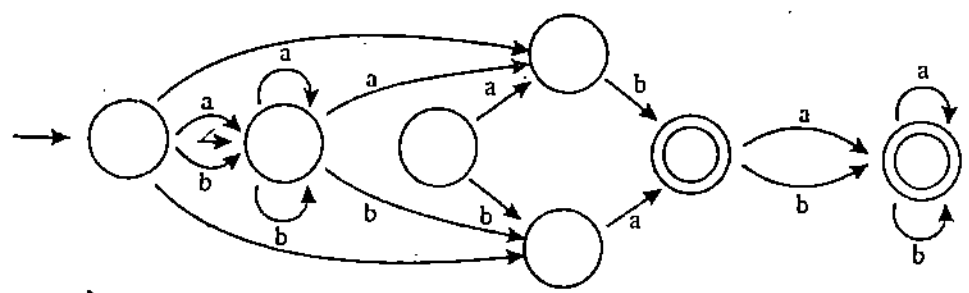


Fig. 32: Removing \wedge

12-1

After minimizing the number of states, we get,

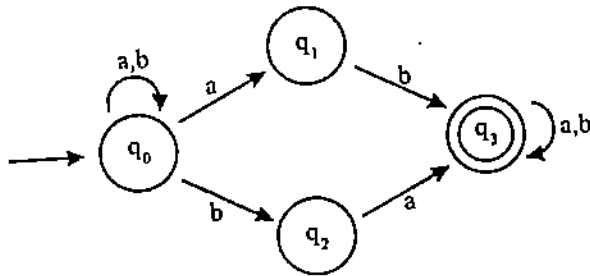


Fig. 33: Equivalent NFA

Step 3: Construction of equivalent DFA.

Table 10

States	Input	
	A	b
$\rightarrow [q_0]$	$[q_0, q_1]$	$[q_0, q_2]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_2, q_3]$
$[q_0, q_2]$	$[q_0, q_2, q_3]$	$[q_0, q_2]$
$[q_0, q_1, q_3]$	$[q_0, q_1, q_3]$	$[q_0, q_2, q_3]$
$[q_0, q_2, q_3]$	$[q_0, q_1, q_3]$	$[q_0, q_2, q_3]$

Diagrammatically it is shown in Figure 34.

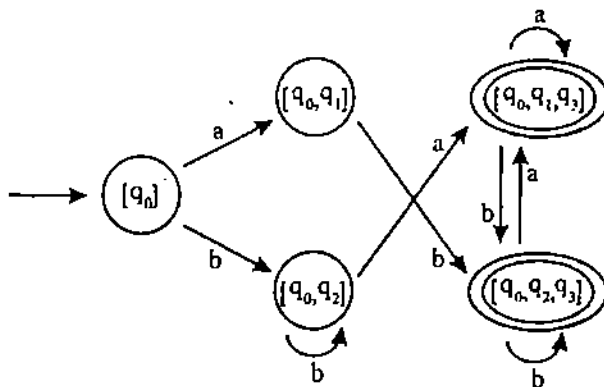


Fig. 34: Equivalent DFA

Now try some exercises.

Ex.10) Find the finite automata equivalent to the following regular expressions:

(i) $ba+(aa+b)a^*b$

(ii) $b+aa+aba^*b$.

(iii) $(a+b) b (a+b)^*$

As you have seen that there exists an equivalent NFA with \wedge -transitions, NFA without \wedge -transitions and DFA to each regular expression. But if there is some transition system, then there exists equivalent regular expression. The algorithm we are going to discuss for this purpose is not restricted to NFA, DFA. This algorithm can be applied to each transition system to find its equivalent regular expression. We convert a transition system to a regular expression by reducing the states. These states are reduced by replacing each state one by one with a corresponding regular expression. The following steps are used:

- If the label is (a, b), then it is replaced by $a+b$.
- First of all, eliminate all the states which are not initial or final states. If we replace the state q_c from the transition given below,

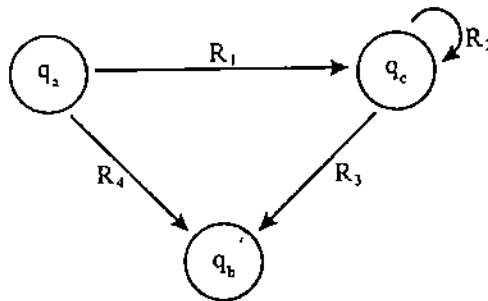


Fig. 35: Transition diagram before elimination of q_c .

then q_c is eliminated by writing its corresponding regular expression $R_1 R_2^* R_3 + R_4$ from q_a to q_b , as follows:

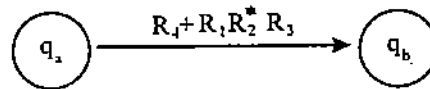


Fig. 36: Transition diagram after elimination of q_c .

Continue the process till only initial and final states remain.

- If initial state is final state and the regular expression is R , such as



Fig. 37: Transition system with the state

the equivalent regular expression is R^*

- If initial state is not final state and is like,

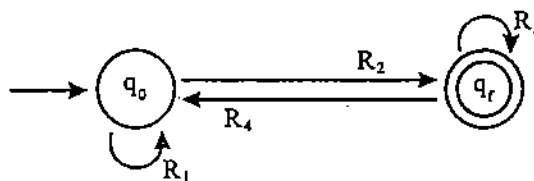


Fig. 38: Transition system with different initial and final state

then this can also be written as

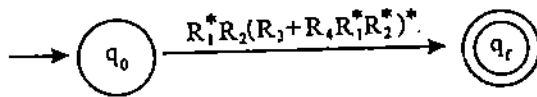


Fig. 39: Showing equivalent regular expression

which is the equivalent regular expression. If these are n final states and $R_1, R_2, R_3, \dots, R_n$ are the regular expressions accepted by these states, then the regular expression accepted by the transition system will be $R_1+R_2+\dots+R_n$.

Now let us try some examples to understand the algorithm well.

Example 9: Find the regular expression equivalent to the given system.

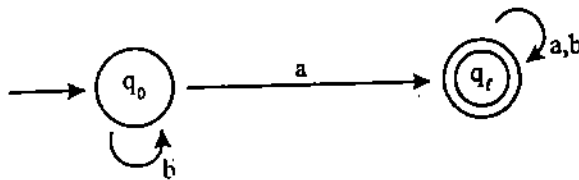


Fig. 40

There is no state which is neither initial nor final so this can be written as

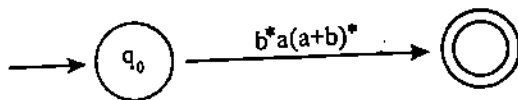


Fig. 41: Equivalent regular expression

The equivalent r.e. is $b^*a(a+b)^*$.

Example 10: Find a regular expression equivalent to

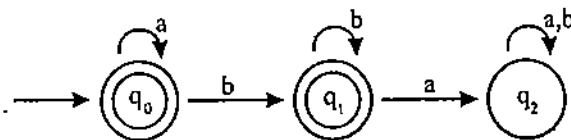


Fig. 42

Firstly, after eliminating q_2 , we get

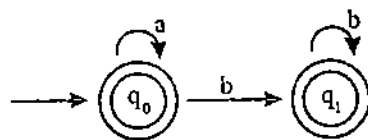


Fig. 43: Equivalent NFA

There are two final states, q_0 and q_1 . The regular expression accepted by q_0 is a^* and the regular expression accepted by q_1 is a^*bb^* . Then, the regular expression accepted by the transition system is

$$a^* + a^*(bb^*) = a^*(\lambda + bb^*) \text{ (distributive property)}$$

$$= a^*b^* \text{ is the equivalent regular expression.}$$

Try some exercise.

Ex.11) Take any regular expression and find the transition system. Using this transition system, find equivalent regular expression and check your result.

2.8 SUMMARY

In this unit we have covered the following:

1. Non-deterministic finite automata.
2. There exist, an equivalent DFA for every NFA.
3. Two Automata M and N are said to be equivalent iff $L(M) = (L(N))$.
4. Pumping lemma with its proof.
5. Application of pumping lemma in establishing a given language as non-regular.
6. Closure properties of regular language and finite automata.
7. Equivalence of regular expression and finite automata. The regular language can be found from a regular expression as well as finite automata. So, these two approaches of regular languages are equivalent.

2.9 SOLUTIONS/ANSWERS

Ex.1) δ is given by

State	Input Σ	
	0	1
$\rightarrow q_0$	q_0, q_1	q_0
q_1	-	q_2
q_2	-	-

$$\delta(q_0, 001) = \delta(q_0, 01) = \delta(q_1, 1) = q_2 \text{ (Accepting state)}$$

$$\begin{aligned} \delta(q_0, 011101) &= \delta(q_0, 11101) \\ &= \delta(q_0, 1101) \\ &= \delta(q_0, 101) \\ &= \delta(q_0, 01) \\ &= \delta(q_1, 1) \end{aligned}$$

$$= (q_2) \text{ (accepting state)}$$

$$\begin{aligned} \delta(q_0, 01110) &= \delta(q_0, 1110) \\ &= \delta(q_0, 110) \\ &= \delta(q_0, 10) \\ &= \delta(q_0, 0) \\ &= q_0 \text{ or } q_1 \text{ (Not an accepting state)} \end{aligned}$$

$$\begin{aligned} \delta(q_0, 010) &= \delta(q_0, 10) \\ &= \delta(q_0, 0) \\ &= q_0 \text{ or } q_1 \text{ (Not an accepting state)} \end{aligned}$$

So, strings 001 and 01110 are accepted by the given automata.

Ex.2)

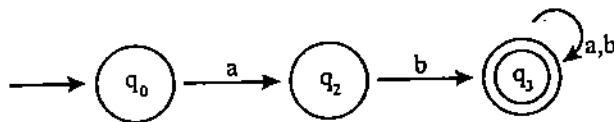


Fig. 44

Ex.3) NFA

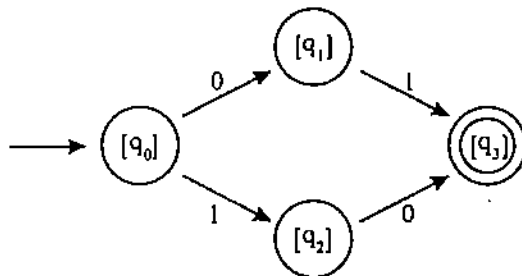


Fig. 45

Transition function is given in the table below:

States	0	1
→ q ₀	q ₁	q ₂
q ₁	-	q ₃
q ₂	-	-
(q ₃)	-	-

Equivalent DFA is

States	0	1
ϕ	ϕ	ϕ
$\rightarrow [q_0]$	$[q_1]$	$[q_2]$
$[q_1]$	$\{\phi\}$	$[q_3]$
$[q_2]$	$[q_3]$	$\{\phi\}$
$[q_3]$	$\{\phi\}$	$\{\phi\}$

Ex.4) DFA is

State	0	1
$\rightarrow [q_1]$	$[q_2, q_3]$	$[q_1]$
$[q_2, q_3]$	$[q_1, q_2]$	$[q_1, q_2]$
$[q_1, q_2]$	$[q_1, q_2, q_3]$	$[q_1]$
$[q_1, q_2, q_3]$	$[q_1, q_2, q_3]$	$[q_1, q_2]$

Ex.5)

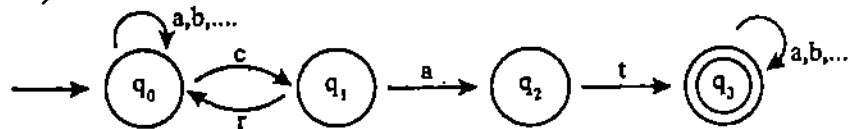


Fig. 46

Ex.7) (i) $L_1 = \{0^m 1^m : m \geq 0\}$

$$w = xyz = 0^m 1^m \Rightarrow |w| = 2m$$

Consider three cases

$$\text{Iff } y = 0^k \Rightarrow w = 0^{m-k} 0^k 1^m$$

$$xy^i z = 0^{m-k} 0^{ik} 1^m$$

$$xy^i z = 0^{m+(i-1)k} 1^m$$

$$\Rightarrow xy^i z \notin L_1 \text{ as } m+(i-1)k \neq m.$$

Similarly, case II with $y = 0^k 1^l$ and case III with $y = 1^k$ can be assumed and will not belong to L_1 .

So, L_1 is not regular

(ii) Similarly, as part (i)

(iii) $L_3 = \{a^p : p \text{ is prime}\}$

Suppose $y = a^m, m > 0$ and $w = xyz = a^p$ so $|w| = p$

$$\Rightarrow xy^i z = xyy^{i-1}z$$

$$\begin{aligned} \Rightarrow |xy^i z| &= |x| + |y| + (i-1)|y| + |z| \\ &= |w| + (i-1)|y| \end{aligned}$$

$$= p + (i-1)|y|$$

$$= p + (i-1)m.$$

If we choose $i-1$ a multiple of p , then we get

$$|xy^iz| = p+kpm$$

$$= (1+km)p$$

which is not a prime number

so, $xy^iz \notin L_3$

(iv) Suppose L is regular, and n be the number of states in automata M .

$w = xyz$ with $|y| \neq 0$, $|xy| \leq n$.

Let us consider $ww = 01^n 01^n \in L_4$

and $|ww| = 2(n+1) > n$.

I case : y has no 0's, i.e., $y = 1^k$; $k \geq 1$

II case : y has only one 0.

Here, y cannot have two 0's. If so $|y| \geq n+2$. But $|y| \leq |xy| \leq n$.

In case I, assume $i = 0$. Then $xy^iz = xz$ and is of the form $01^m 01^n$, where $m = n - k < n$ or of the form $01^n 01^n$. These both values cannot be written in form of ww with $w \in \{0, 1\}^*$ and so $xz \notin L$. In case II also, take $i = 0$ then 0 will be removed and $xz = 01^n 1^n$ again this cannot be written in ww form. Thus, in both the cases we get a contradiction. Therefore, L is not regular.

(v) Left as an exercise.

Ex.8) Any example of a language may be given which is not regular. Use again pumping lemma to justify

- Ex.9) 1. ϕ
 2. $(a^*)a$
 3. ϕ
 4. $(a+b)^* aa$
 5. $(aa+ab+ba+bb)^*$

Ex.10) (i) Equivalent NFA is

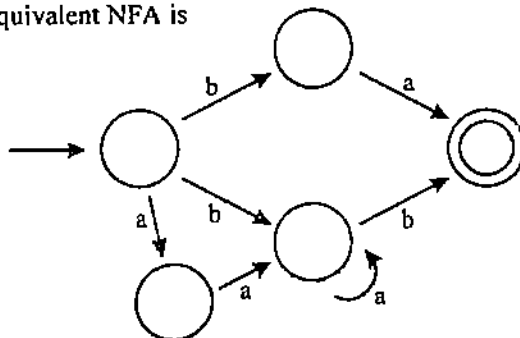


Fig. 47

(ii) NFA is

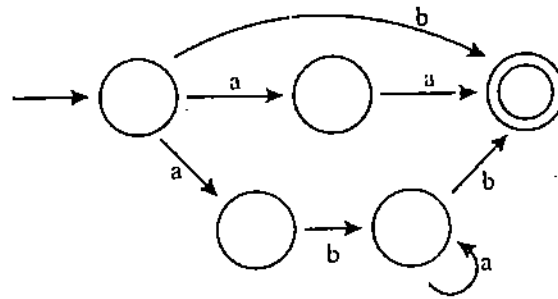


Fig. 48

UNIT 3 CONTEXT FREE GRAMMAR

Structure	Page Nos.
3.0 Introduction	53
3.1 Objectives	53
3.2 Grammar and its Classification	53
3.3 Context Free Grammar (CFG)	60
3.4 Pushdown Automata (PDA)	64
3.5 Non-Context Free Languages, Pumping Lemma for CFL	67
3.6 Equivalence of Context free Grammar and Push Down Automata	73
3.7 Summary	77
3.8 Solutions/Answers	78

3.0 INTRODUCTION

In unit 2, we studied the class of regular languages and their representations through regular expressions and finite automata. We have also seen that not all languages are regular. If a language is not regular then there should be other categories of language also. We have also seen that languages are defined by regular expression. Regular languages are closed under union, product, Kleene star, intersection and complement. Application areas are: text editors, sequential circuits, etc. The corresponding acceptor is Finite Automata.

Now, we shall discuss the concept of context free grammar for a larger class of languages. Language will be defined by context free grammar. Corresponding acceptor is Pushdown Automata. In this unit we shall check whether a context free language is closed under union, product and Kleene star or not. Language that will be defined by context free grammar is context free language. Application areas are: programming languages, statements and compilers.

3.1 OBJECTIVES

After studying this unit, you should be able to

- create a grammar from language and vice versa;
- explain and create context free grammar and language;
- define the pushdown automata;
- apply the pumping lemma for non-context free languages; and
- find the equivalence of context free grammar and Pushdown Automata

In unit 1, we discussed language and a regular language. A language is meaningful if a grammar is used to derive the language. So, it is very important to construct a language from a grammar. As you know all languages are not regular. This non-regular languages are further categorised on the basis of classification of grammar.

3.2 GRAMMAR AND ITS CLASSIFICATION

In our day-to-day life, we often use the common words such as grammar and language. Let us discuss it through one example.

Example 1: If we talk about a sentence in English language. "Ram reads", this sentence is made up of Ram and reads. Ram and reads are replaced for <noun> and

<verb>. We can say simply that a sentence is changed by noun and verb and is written as

$$\langle \text{sentence} \rangle \rightarrow \langle \text{noun} \rangle \langle \text{verb} \rangle$$

where noun can be replaced with many such values as Ram, Sam, Gita.... and also <verb> can be replaced with many other values such as read, write, go As noun and verb are replaced, we easily write

<noun>	→	I
<noun>	→	Ram
<noun>	→	Sam
<verb>	→	reads
<verb>	→	writes

From the above, we can collect all the values in two categories. One is with the parameter changing its values further, and another is with termination. These collections are called variables and terminals, respectively. In the above discussion variables are, <sentence>, <noun> and <verb>, and terminals are I, Ram, Sam, read, write. As the sentence formation is started with <sentence>, this symbol is special symbol and is called start symbol.

Now formally, a Grammar $G = (V, \Sigma, P, S)$ where,

- V is called the set of variables. e.g., $\{S, A, B, C\}$
- Σ is the set of terminals, e.g. $\{a, b\}$
- P is a set of production rules
(- Rules of the form $A \rightarrow \alpha$ where $A \in (V \cup \Sigma)^+$ and $\alpha \in (V \cup \Sigma)^+$ e.g., $S \rightarrow aA$).
- S is a special variable called the start symbol $S \in V$.

Structure of grammar: If L is a language over an alphabet A , then a grammar for L consists of a set of grammar rules of the form

$$x \rightarrow y$$

where x and y denote strings of symbols taken from A and from a set of grammar symbols disjoint from A . The grammar rule $x \rightarrow y$ is called a production rule, and application of production rule (x is replaced by y), is called derivation.

Every grammar has a special grammar symbol called the start symbol and there must be at least one production with the left side consisting of only the start symbol. For example, if S is the start symbol for a grammar, then there must be at least one production of the form $S \rightarrow y$.

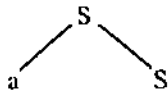
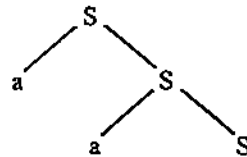
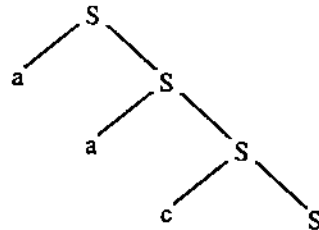
Example 2: Suppose $A = \{a, b, c\}$ then a grammar for the language A^* can be described by the following four productions:

- $S \rightarrow \wedge$ (i)
- $S \rightarrow aS$ (ii)
- $S \rightarrow bS$ (iii)
- $S \rightarrow cS$ (iv)

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aa\wedge S \Rightarrow aacbS \Rightarrow aacb = aacb$$

using prod.(u) using prod.(ii) using prod.(iv) using prod.(iii) using prod.(i)

The desired derivation of the string is aacb. Each step in a derivation corresponds to a branch of a tree and this tree is called **parse tree**, whose root is the start symbol. The completed derivation and parse tree are shown in the Figure 1,2,3:

Fig. 1: $S \Rightarrow aS$ Fig. 2: $S \Rightarrow aS \Rightarrow aaS$ Fig. 3: $S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS$

Let us derive the string aacb, its parse tree is shown in figure 4.

$S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS \Rightarrow aacbS \Rightarrow aacb\Lambda = aacb$

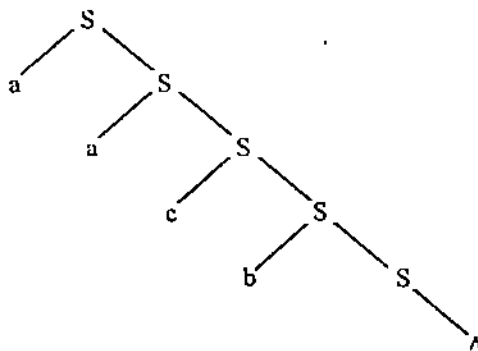


Fig. 4: Parse tree deriving aacb

Sentential Form: A string made up of terminals and/or non-terminals is called a sentential form.

In example 1, formally grammar is rewritten as

In $G = (V, \Sigma, P, S)$ where

$V = \{ \langle \text{sentence} \rangle, \langle \text{noun} \rangle, \langle \text{verb} \rangle \}$

$\Sigma = \{ \text{Ram, reads, ...} \}$

$P = \langle \text{sentence} \rangle \rightarrow \langle \text{noun} \rangle \langle \text{verb} \rangle$

$\langle \text{noun} \rangle \rightarrow \text{Ram}$

$\langle \text{verb} \rangle \rightarrow \text{reads, and}$

$S = \langle \text{sentence} \rangle$

If x and y are sentential forms and $\alpha \rightarrow \beta$ is a production, then the replacement of α by β in $x\alpha y$ is called a derivation, and we denote it by writing

$$x\alpha y \Rightarrow x\beta y$$

To the left hand side of the above production rule x is left context and y is right context. If the derivation is applied to left most variable of the right hand side of any production rule, then it is called leftmost derivation. And if applied to rightmost then is called rightmost derivation.

The language of a Grammar :

A language is generated from a grammar. If G is a grammar with start symbol S and set of terminals Σ , then the language of G is the set

$$L(G) = \{W \mid W \in \Sigma^* \text{ and } S \xRightarrow{G} W\}.$$

Any derivation involves the application production Rules. If the production rule is applied once, then we write $\alpha \xRightarrow{G} B$. When it is more than one, it is written as $\alpha \xRightarrow{G} \beta$

Recursive productions: A production is called recursive if its left side occurs on its right side. For example, the production $S \rightarrow aS$ is recursive. A production $A \rightarrow \alpha$ is indirectly recursive. If A derives a sentential form that contains A . Then, suppose we have the following grammar:

$$\begin{aligned} S &\rightarrow b/aA \\ A &\rightarrow c/bS \end{aligned}$$

the productions $S \rightarrow aA$ and $A \rightarrow bs$ are both indirectly recursive because of the following derivations:

$$\begin{aligned} S &\Rightarrow aA \Rightarrow abS, \\ A &\Rightarrow bS \Rightarrow baA \end{aligned}$$

A grammar is recursive if it contains either a recursive production or an indirectly recursive production.

A grammar for an infinite language must be recursive.

Example 3: Consider $\{\Lambda, a, aa, \dots, a^n, \dots\} = \{a^n \mid n \geq 0\}$.

Notice that any string in this language is either Λ or of the form ax for some string x in the language. The following grammar will derive any of these strings:

$$S \rightarrow \Lambda/aS.$$

Now, we shall derive the string aaa :

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaa.$$

Example 4: Consider $\{\Lambda, ab, aabb, \dots, a^n b^n, \dots\} = \{a^n b^n \mid n \geq 0\}$.

Notice that any string in this language is either Λ or of the form axb for some string x in the language. The following grammar will derive any of the strings:

$$S \rightarrow \Lambda/aSb.$$

For example, we will derive the string $aaabbb$;

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb.$$

Example 5: Consider a language $\{\Lambda, ab, abab, \dots, (ab)^n, \dots\} = \{(ab)^n \mid n \geq 0\}$.

Notice that any string in this language is either Λ or of the form abx for some string x in the language. The following grammar will derive any of these strings:

$$S \rightarrow \Lambda / abS.$$

For example, we shall derive the string $ababab$:

$$S \Rightarrow abS \Rightarrow ababS \Rightarrow abababS \Rightarrow ababab.$$

Sometimes, a language can be written in terms of simpler languages, and a grammar can be constructed for the language in terms of the grammars for the simpler languages. We will now concentrate on operations of union, product and closure.

Suppose M and N are languages whose grammars have disjoint sets of non-terminals. Suppose also that the start symbols for the grammars of M and N are A and B , respectively. Then, we use the following rules to find the new grammars generated from M and N :

Union Rule: The language $M \cup N$ starts with the two productions

$$S \rightarrow A/B.$$

Product Rule: The language MN starts with the production.

$$S \rightarrow AB$$

Closure Rule: The language M^* starts with the production

$$S \rightarrow AS/\Lambda.$$

Example 6: Using the Union Rule:

Let's write a grammar for the following language:

$$L = \{\Lambda, a, b, aa, bb, \dots, a^n, b^n, \dots\}.$$

L can be written as union.

$$L = M \cup N,$$

Where $M = \{a^n \mid n \geq 0\}$ and $N = \{b^n \mid n \geq 0\}$.

Thus, we can write the following grammar for L :

$$\begin{aligned} S &\rightarrow A \mid B \text{ union rule,} \\ A &\rightarrow \Lambda/aA \text{ grammar for } M, \\ B &\rightarrow \Lambda/bB \text{ grammar for } N. \end{aligned}$$

Example 7: Using the Product Rule:

We shall write a grammar for the following language :

$$L = \{a^m b^n \mid m, n \geq 0\}.$$

L can be written as a product $L = MN$, where $M = \{a^m \mid m \geq 0\}$ and $N = \{b^n \mid n \geq 0\}$. Thus we can write the following grammar for L :

$$\begin{aligned}
 S &\rightarrow AB \text{ product rule} \\
 A &\rightarrow \wedge/aA \text{ grammar for } M, \\
 B &\rightarrow \wedge/bB \text{ grammar for } N,
 \end{aligned}$$

Example 8: - Using the Closure Rule: For the language L of all strings with zero or more occurrence of aa or bb. $L = \{aa, bb\}^*$. If we let $M = \{aa, bb\}$, then $L = M^*$. Thus, we can write the following grammar for L:

$$\begin{aligned}
 S &\rightarrow AS/\wedge \text{ closure rule,} \\
 A &\rightarrow aa/bb \text{ grammar for } M.
 \end{aligned}$$

We can simplify the grammar by substituting for A to obtain the following grammar:

$$S \rightarrow aaS/bbS/\wedge$$

Example 9: Let $\Sigma = \{a, b, c\}$. Let S be the start symbol. Then, the language of palindromes over the alphabet Σ has the grammar.

$$S \rightarrow aSa/bSb/cSc/a/b/c/\wedge.$$

For example, the palindrome abcba can be derived as follows:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abcba$$

Ambiguity: A grammar is said to be ambiguous if its language contains some string that has two different parse tree. This is equivalent to saying that some string has two distinct leftmost derivations or that some string has two distinct rightmost derivations.

Example 10: Suppose we define a set of arithmetic expressions by the grammar:

$$E \rightarrow a/b/E-E$$

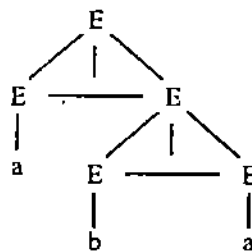


Fig. 5: Parse Tree

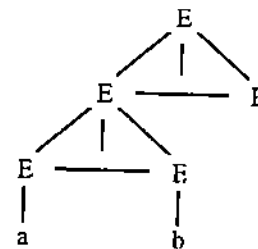


Fig. 6: Parse Tree showing ambiguity

This is the parse tree for an ambiguous string.

The language of the grammar $E \rightarrow a/b/E-E$ contains strings like a, b, b-a, a-b-a, and b-b-a-b. This grammar is ambiguous because it has a string, namely, a-b-a, that has two distinct parse trees.

Since having two distinct parse trees mean the same as having two distinct left most derivations.

$$E \Rightarrow E-E \Rightarrow a-E \Rightarrow a-E-E \Rightarrow a-b-E \Rightarrow a-b-a.$$

$$E \Rightarrow E-E \Rightarrow E-E-E \Rightarrow a-E-E \Rightarrow a-b-E \Rightarrow a-b-a.$$

The same is the case with rightmost derivation.

- A derivation is called a **leftmost derivation** if at each step the leftmost non-terminal of the sentential form is reduced by some production.
- A derivation is called a **rightmost derivation** if at each step the rightmost non-terminal of the sentential form is reduced by some production.

Let us try some exercises.

Ex.1) Given the following grammar

$$S \rightarrow S[S]/\wedge$$

For each of the following strings, construct a leftmost derivation, a rightmost derivation and a parse tree.

- (a) $[]$ (b) $[[]]$ (c) $[] []$ (d) $[[] []]$

Ex.2) Find a grammar for each language

- (a) $\{a^m b^n \mid m, n \in \mathbb{N}, n > m\}$.
 (b) $\{a^n b c^n \mid n \in \mathbb{N}\}$.

Ex.3) Find a grammar for each language:

- (a) The even palindromes over $\{a, b\}$.
 (b) The odd palindromes over $\{a, b\}$.

Chomsky Classification for Grammar:

As you have seen earlier, there may be many kinds of production rules. So, on the basis of production rules we can classify a grammar. According to Chomsky classification, grammar is classified into the following types:

Type 0: This grammar is also called **unrestricted grammar**. As its name suggests, it is the grammar whose production rules are unrestricted.

All grammars are of type 0.

Type 1: This grammar is also called **context sensitive grammar**. A production of the form $xAy \rightarrow xcy$ is called a type 1 production if $\alpha \neq \wedge$, which means length of the working string does not decrease.

In other words, $|xAy| \leq |xcy|$ as $\alpha \neq \wedge$. Here, x is left context and y is right context.

A grammar is called type 1 grammar, if all of its productions are of type 1. For this, grammar $S \rightarrow \wedge$ is also allowed.

The language generated by a type 1 grammar is called a type 1 or **context sensitive language**.

Type 2: The grammar is also known as **context free grammar**. A grammar is called type 2 grammar if all the production rules are of type 2. A production is said to be of type 2 if it is of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^+$. In other words, the left hand side of production rule has no left and right context. The language generated by a type 2 grammar is called **context free language**.

Type 3: A grammar is called type 3 grammar if all of its production rules are of type 3. (A production rule is of type 3 if it is of form $A \rightarrow \wedge$, $A \rightarrow a$ or $A \rightarrow aB$ where $a \in \Sigma$ and $A, B \in V$), i.e., if a variable derives a terminal or a terminal with one variable. This type 3 grammar is also called **regular grammar**. The language generated by this grammar is called **regular language**.

Ex.4) Find the highest type number that can be applied to the following grammar:

- (a) $S \rightarrow ASB/b, A \rightarrow aA$
- (b) $S \rightarrow aSa/bSb/a/b/\wedge$
- (c) $S \rightarrow Aa, A \rightarrow S/Ba, B \rightarrow abc.$

3.3 CONTEXT FREE GRAMMAR

We know that there are non-regular languages. For example: $\{a^n b^n \mid n \geq 0\}$ is non-regular language. Therefore, we can't describe the language by any of the four representations of regular languages, regular expressions, DFAs, NFAs, and regular grammars.

Language $\{a^n b^n \mid n \geq 0\}$ can be easily described by the non-regular grammar:

$$S \rightarrow \wedge / aSb.$$

So, a context-free grammar is a grammar whose productions are of the form :

$$S \rightarrow x$$

Where S is a non-terminal and x is any string over the alphabet of terminals and non-terminals. Any regular grammar is context-free. A language is context-free language if it is generated by a context-free grammar.

A grammar that is not context-free must contain a production whose left side is a string of two or more symbols. For example, the production $Sc \rightarrow x$ is not part of any context-free grammar.

Most programming languages are context-free. For example, a grammar for some typical statements in an imperative language might look like the following, where the words in bold face are considered to be the single terminals:

$$S \rightarrow \text{while } E \text{ do } S / \text{if } E \text{ then } S \text{ else } S / \{SL\} / I = E$$

$$L \rightarrow SL / \wedge$$

$$E \rightarrow \dots (\text{description of an expression})$$

$$I \rightarrow \dots (\text{description of an identifier}).$$

We can combine context-free languages by union, language product, and closure to form new context-free languages.

Definition: A context-free grammar, called a CFG, consists of three components:

1. An alphabet Σ of letters called **terminals** from which we are going to make strings that will be the words of a language.
2. A set of symbols called **non-terminals**, one of which is the symbols, start symbol.

3. A finite set of productions of the form

One non-terminal \rightarrow finite string of terminals and/or non-terminals.

Where the strings of terminals and non-terminals can consist of only terminals or of only non-terminals, or any combination of terminals and non-terminals or even the empty string.

The language generated by a CFG is the set of all strings of terminals that can be produced from the start symbol S using the productions as substitutions. A language generated by a CFG is called a context-free language.

Example 11: Find a grammar for the language of decimal numerals by observing that a decimal numeral is either a digit or a digit followed by a decimal numeral.

$$\begin{aligned} S &\rightarrow D/DS \\ D &\rightarrow 0/1/2/3/4/5/6/7/8/9 \end{aligned}$$

$$S \Rightarrow DS \Rightarrow 7S \Rightarrow 7DS \Rightarrow 7DDS \Rightarrow 78DS \Rightarrow 780S \Rightarrow 780D \Rightarrow 780.$$

Example 12: Let the set of alphabet $A = \{a, b, c\}$

Then, the language of palindromes over the alphabet A has the grammar:

$$S \rightarrow aSa \mid bSb \mid cSc \mid a \mid b \mid c \mid \Lambda$$

For example, the palindrome $abcba$ can be derived as follows:

$$P \Rightarrow aPa \Rightarrow abPba \Rightarrow abcba$$

Example 13: Let the CFG is $S \rightarrow L \mid LA$

$$\begin{aligned} A &\rightarrow LA \mid DA \mid \Lambda \\ L &\rightarrow a \mid b \mid \dots \mid Z \\ D &\rightarrow 0 \mid 1 \mid \dots \mid 9 \end{aligned}$$

The language generated by the grammar has all the strings formed by $a, b, c, \dots, z, 0, 1, \dots, 9$.

We shall give a derivation of string $a2b$ to show that it is an identifier.

$$S \Rightarrow LA \Rightarrow aA \Rightarrow aDA \Rightarrow a2A \Rightarrow a2LA \Rightarrow a2bA \Rightarrow a2b$$

Context-Free Language: Since the set of regular language is closed under all the operations of union, concatenation, Kleen star, intersection and complement. The set of context free languages is closed under union, concatenation, Kleen star only.

Union

Theorem 1: if L_1 and L_2 are context-free languages, then $L_1 \cup L_2$ is a context-free language.

Proof: If L_1 and L_2 are context-free languages, then each of them has a context-free grammar; call the grammars G_1 and G_2 . Our proof requires that the grammars have no non-terminals in common. So we shall subscript all of G_1 's non-terminals with a 1 and subscript all of G_2 's non-terminals with a 2. Now, we combine the two grammars into one grammar that will generate the union of the two languages. To do this, we add one new non-terminal, S , and two new productions.

$$S \rightarrow S_1 \\ | S_2$$

S is the starting non-terminal for the new union grammar and can be replaced either by the starting non-terminal for G_1 or for G_2 , thereby generating either a string from L_1 or from L_2 . Since the non-terminals of the two original languages are completely different, and once we begin using one of the original grammars, we must complete the derivation using only the rules from that original grammar. Note that there is no need for the alphabets of the two languages to be the same.

Concatenation

Theorem 2: If L_1 and L_2 are context-free languages, then L_1L_2 is a context-free language.

Proof: This proof is similar to the last one. We first subscript all of the non-terminals of G_1 with a 1 and all the non-terminals of G_2 with a 2. Then, we add a new nonterminal, S, and one new rule to the combined grammar:

$$S \rightarrow S_1S_2$$

S is the starting non-terminal for the concatenation grammar and is replaced by the concatenation of the two original starting non-terminals.

Kleene Star

Theorem 3: If L is a context-free language, then L^* is a context-free language.

Proof: Subscript the non-terminals of the grammar for L with a 1. Then add a new starting nonterminal, S, and the rules

$$S \rightarrow S_1S \\ | \Lambda$$

The rule $S \rightarrow S_1S$ is used once for each string of L that we want in the string of L^* , then the rule $S \rightarrow \Lambda$ is used to kill off the S.

Intersection

Now, we will show that the set of context-free languages is not closed under intersection. Think about the two languages $L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ and $L_2 = \{a^m b^n c^n \mid n, m \geq 0\}$. These are both context-free languages and we can give a grammar for each one:

G_1 :

$$S \rightarrow AB \\ A \rightarrow aAb \\ | \Lambda \\ B \rightarrow cB \\ | \Lambda$$

G_2 :

$$S \rightarrow AB \\ A \rightarrow aA \\ | \Lambda \\ B \rightarrow bBc \\ | \Lambda$$

The strings in L_1 contain the same number of a's as b's, while the strings in L_2 contain the same number of b's as c's. Strings that have to be both in L_1 and in L_2 , i.e., strings in the intersection, must have the same numbers of a's as b's and the same number of b's as c's.

Thus, $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$. Using Pumping lemma for context-free languages it can be proved easily that $\{a^n b^n c^n \mid n \geq 0\}$ is not context-free language. So, the class of context-free languages is not closed under intersection.

Although the set is not closed under intersection, there are cases in which the intersection of two context-free languages is context-free. Think about regular languages, for instance. All regular languages are context-free, and the intersection of two regular languages is regular. We have some other special cases in which an intersection of two context-free languages is context, free.

Suppose that L_1 and L_2 are context-free languages and that $L_1 \subseteq L_2$. Then $L_2 \cap L_1 = L_1$ which is a context-free language. An example is $\text{EQUAL} \cap \{a^n b^n\}$. Since strings in $\{a^n b^n\}$ always have the same number of a's as b's, the intersection of these two languages is the set $\{a^n b^n\}$, which is context-free.

Another special case is the intersection of a regular language with a non-regular context-free language. In this case, the intersection will always be context-free. An example is the intersection of $L_1 = a^* b^* a^*$, which is regular, with $L_2 = \text{PALINDROME}$. $L_1 \cap L_2 = \{a^m b^n a^m \mid m, n \geq 0\}$. This language is context-free.

Complement

The set of context-free languages is not closed under complement, although there are again cases in which the complement of a context-free language is context-free.

Theorem 4: The set of context-free languages is not closed under complement.

Proof: Suppose the set is closed under complement. Then, if L_1 and L_2 are context-free, so are L_1' and L_2' . Since the set is closed under union, $L_1 \cup L_2$ is also context-free, as is $(L_1 \cup L_2)'$. But, this last expression is equivalent to $L_1' \cap L_2'$ which is not guaranteed to be context-free. So, our assumption must be incorrect and the set is not closed under complement.

Here is an example of a context-free language whose complement is not context-free. The language $\{a^n b^n c^n \mid n \geq 1\}$ is not context-free, but the author proves that the complement of this language is the union of seven different context-free languages and is thus context-free. Strings that are not in $\{a^n b^n c^n \mid n \geq 1\}$ must be in one of the following languages:

1. $M_{pq} = \{a^p b^q c^r \mid p, q, r \geq 1 \text{ and } p > q\}$ (more a's than b's)
2. $M_{qp} = \{a^p b^q c^r \mid p, q, r \geq 1 \text{ and } q > p\}$ (more b's than a's)
3. $M_{pr} = \{a^p b^q c^r \mid p, q, r \geq 1 \text{ and } s > r\}$ (more a's than c's)
4. $M_{rp} = \{a^p b^q c^r \mid p, q, r \geq 1 \text{ and } r > p\}$ (more c's than a's)
5. $M =$ the complement of $a^+ b^+ c^+$ (letters out of order)

Using Closure Properties

Sometimes, we can use closure properties to prove that a language is not context-free. Consider the language our author calls $\text{DOUBLEWORD} = \{ww \mid w \in (a+b)^*\}$. Is this language context-free? Assume that it is. Form the intersection of DOUBLEWORD with the regular language $a^+ b^+ a^+ b^+$, we know that the intersection of a context-free language and a regular language is always context-free. The intersection of

DOUBLEWORD and is $a^n b^m a^n b^m \mid n, m \geq 1$. But, this language is not context-free, so DOUBLEWORD cannot be context-free.

Think carefully when doing unions and intersections of languages if one is a superset of the other. The union of PALINDROME and $(a+b)^*$ is $(a+b)^*$, which is regular. So, sometimes the union of a context-free language and a regular language is regular. The union of PALINDROME and a^* is PALINDROME, which is context-free but not regular.

Now try some exercises:

Ex.5) Find CFG for the language over $\Sigma = \{a,b\}$.

(a) All words of the form

$$a^x b^y a^z, \text{ where } x, y, z = 1, 2, 3, \dots \text{ and } y = 5x + 7z$$

(b) For any two positive integers p and q , the language of all words of the form $a^x b^y a^z$, where $x, y, z = 1, 2, 3, \dots$ and $y = px + qz$.

3.4 PUSHDOWN AUTOMATA (PDA)

Informally, a pushdown automata is a finite automata with stack. The corresponding acceptor of context-free grammar is pushdown automata. There is one start state and there is a possibly empty-set of final states. We can imagine a pushdown automata as a machine with the ability to read the letters of an input string, perform stack operations, and make state changes.

The execution of a PDA always begins with one symbol on the stack. We should always specify the initial symbol on the stack. We assume that a PDA always begins execution with a particular symbol on the stack. A PDA will use three stack operations as follows:

- (i) The **pop** operation reads the top symbol and removes it from the stack.
- (ii) The **push** operation writes a designated symbol onto the top of the stack. For example, push (x) means put x on top of the stack.
- (iii) The **nop** does nothing to the stack.

We can represent a pushdown automata as a finite directed graph in which each state (i.e., node) emits zero or more labelled edges. Each edge from state i to state j labelled with three items as shown in the Figure 7, where L is either a letter of an alphabet or \wedge , S is a stack symbol, and O is the stack operation to be performed.

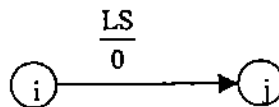


Fig. 7: Directed graph

It takes fine pieces of information to describe a labelled edge. We can also represent it by the following 5-tuple, which is called a PDA instruction.

$$(i, L, S, O, j)$$

An instruction of this form is executed as follows, where w is an input string whose letters are scanned from left to right.

If the PDA is in state i , and either L is the current letter of w being scanned or $L = \wedge$, and the symbol on top of the stack is S , then perform the following actions:

- (1) execute the stack operation 0;
- (2) move to the state j ; and
- (3) if $L \neq \Lambda$, then scan right to the next letter of w .

A string is accepted by a PDA if there is some path (i.e., sequence of instructions) from the start state to the final state that consumes all letters of the string. Otherwise, the string is rejected by the PDA. The language of a PDA is the set of strings that it accepts.

Nondeterminism: A PDA is deterministic if there is at most one move possible from each state. Otherwise, the PDA is non-deterministic. There are two types of non-determinism that may occur. One kind of non-determinism occurs exactly when a state emits two or more edges labelled with the same input symbol and the same stack symbol. In other words, there are two 5-tuples with the same first three components. For example, the following two 5-tuples represent nondeterminism:

$$(i, b, c, \text{pop}, j)$$

$$(i, b, c, \text{push}(D), k).$$

The second kind of nondeterminism occurs when a state emits two edges labelled with the same stack symbol, where one input symbol is Λ and the other input symbol is not. For example, the following two 5-tuples represent non-determinism because the machine has the option of consuming the input letter b or cleaning it alone.

$$(i, \Lambda, c, \text{pop}, j)$$

$$(i, b, c, \text{push}(D), k).$$

Example 14: The language $\{a^n b^n \mid n \geq 0\}$ can be accepted by a PDA. We will keep track of the number of a 's in an input string by pushing the symbol Y onto the stack for each a . A second state will be used to pop the stack for each b encountered. The following PDA will do the job, where x is the initial symbol on the stack:

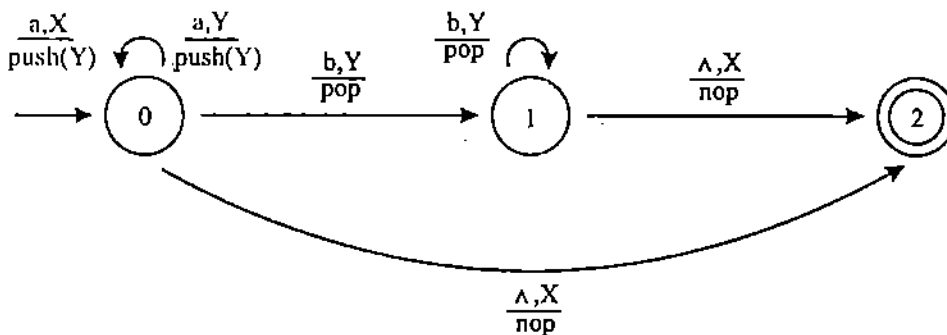


Fig. 8: Pushdown Automata

The PDA can be represented by the following six instructions:

$$(0, \Lambda, X, \text{pop}, 2)$$

$$(0, a, X, \text{push}(Y), 0),$$

$$(0, a, Y, \text{push}(Y), 0),$$

$$(0, b, Y, \text{pop}, 1),$$

$$(1, b, Y, \text{pop}, 1),$$

$$(1, \Lambda, X, \text{pop}, 2).$$

This PDA is non-deterministic because either of the first two instructions in the list can be executed if the first input letter is a and X is on the top of the stack. A computation sequence for the input string aabb can be written as follows:

- (0, aabb, X) start in state 0 with X on the stack.
- (0, abb, YX) consume a and push Y,
- (0, bb, YYX) consume a and push Y,
- (1, b, YX) consume b and pop.
- (0, ^, X) consume b and pop.
- (2, ^, X) move to the final state.

Equivalent Forms of Acceptance:

Above, we defined acceptance of a string by a PDA in terms of final state acceptance. That is a string is accepted if it has been consumed and the PDA is in a final state. But, there is an alternative definition of acceptance called empty stack acceptance, which requires the input string to be consumed and the stock to be empty, with no requirement that the machine be in any particular state. The class of languages accepted by PDAs that use empty stack acceptance is the same class of languages accepted by PDAs that use final state acceptance.

Example 15: (An empty stack PDA): Let's consider the language $\{a^n b^n \mid n \geq 0\}$, the PDA that follows will accept this language by empty stack, where X is the initial symbol on the stack.

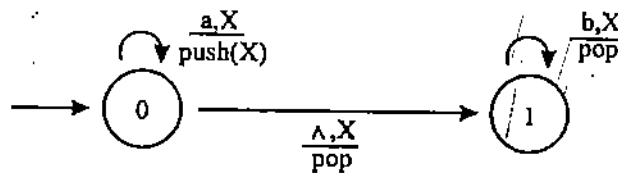


Fig. 9: Pushdown Automata

PDA shown in figure 9 can also be represented by the following three instructions:

- (0, a, X, push (X), 0),
- (0, ^, X, pop, 1),
- (1, b, X, pop, 1).

This PDA is non-deterministic. Let's see how a computation proceeds. For example, a computation sequence for the input string aabb can be as follows:

- (0, aabb, X) start in state 0 with X on the stack
- (0, abb, XX) consume a and push X
- (0, bb, XXX) consume a and push X
- (1, bb, XX) pop.
- (1, b, X) consume b and pop
- (1, ^, ^) consume b and pop (stack is empty)

Now, try some exercises.

Ex.6) Build a PDA that accepts the language odd palindrome.

Ex.7) Build a PDA that accepts the language even palindrome.

3.5 NON-CONTEXT FREE LANGUAGES

Every context free grammar can always be represented in a very interesting form. This form is known as **Chomsky Normal Form (CNF)**.

A context-free grammar is said to be in Chomsky Normal Form if the right hand side of each production has either a terminal or two variables as $S \rightarrow a$, $S \rightarrow AB$ and $S \rightarrow \Lambda$ if $\Lambda \in L(G)$. If $\Lambda \in L(G)$, then S should not appear to the right hand side of any production. To construct a CFG in CNF we can develop a method. In CFG, $S \rightarrow a$ is already allowed, if the production is of form $S \rightarrow aA$ then can be replaced with $S \rightarrow BA$ and $B \rightarrow a$ in CNF. If the production is of the form $S \rightarrow ABC$, it can be written as $S \rightarrow AD$ and $D \rightarrow BC$. Using these simple methods, every CFG can be constructed in CNF.

Example 16: Reduce the following grammar, into CNF.

- (i) $S \rightarrow aAB, A \rightarrow aE/bAE, E \rightarrow b, B \rightarrow d$
 (ii) $S \rightarrow A0B, A \rightarrow AA/0S/0, B \rightarrow 0BB/1S/1$

Solution: (i) $S \rightarrow aAB$ is rewritten in CNF as $S \rightarrow FG, F \rightarrow a$ and $G \rightarrow AB$
 $A \rightarrow aE$ is rewritten as $A \rightarrow FE$ in CNF.
 $A \rightarrow bAE$ in CNF is $A \rightarrow HI, H \rightarrow b$ and $I \rightarrow AE$.

So Chomsky Normal Form of CFG is

$S \rightarrow FG, F \rightarrow a, G \rightarrow AB, A \rightarrow FE, A \rightarrow HI,$
 $H \rightarrow b, I \rightarrow AE, E \rightarrow b$ and $B \rightarrow d$

(ii) left as an exercise.

In this section, we will prove that not all languages are context-free. Any context-free grammar can be put into Chomsky Normal Form. Here is our first theorem.

Theorem 5: Let G be a grammar in Chomsky Normal Form. Call the productions that have two non-terminals on the righthand side live productions and call the ones that have only a terminal on the right-hand side dead productions. If we are restricted to using the live productions of the grammar at most once each, we can generate only a finite number of words.

Proof: Each time when we use a live production, we increase the number of non-terminals in a working string by one. Each time when we use a dead production, we decrease the number of non-terminals by one. In a derivation starting with non-terminal S and ending with a string of terminals, we have to apply one more dead production than live production.

Suppose G has p live productions. Any derivation that does not reuse a live production can use at most p live and $p+1$ dead productions. Each letter in the final string results from one dead production, so words produced without reusing a live production must have no more than $p+1$ letters. There are a finite number of such words.

When doing a leftmost derivation, we replace the leftmost non-terminal at every step. If the grammar is in Chomsky Normal Form, each working string in a leftmost derivation is made up of a group of terminals followed by a group non-terminals. Such working strings are called *leftmost Chomsky working strings*.

Suppose we use a live production $Z \rightarrow XY$ twice in the derivation of some word w . Before the first use of $Z \rightarrow XY$ the working string has the form s_1Zs_2 where s_1 is a string of terminals and s_2 is a string of nonterminals. Before the second use of $Z \rightarrow XY$ the working string has form $s_1s_3Zs_4$ where s_3 is a string of terminals and s_4 is a string of non-terminals.

Suppose we draw a derivation tree representing the leftmost derivation in which we use $Z \rightarrow XY$ twice. The second Z we add to the tree could be a descendant of the first Z or it could come from some other nonterminal in s_2 . Here are examples illustrating the two cases:

Case 1: Z is a descendant of itself.

$S \rightarrow AZ$
 $Z \rightarrow BB$
 $B \rightarrow ZA$
 | b
 $A \rightarrow a$

Beginning of a leftmost derivation:

$S \Rightarrow AZ$
 $\Rightarrow aB$
 $\Rightarrow aBB$
 $\Rightarrow abB$
 $\Rightarrow abZA$

Derivation tree is shown in figure 10:

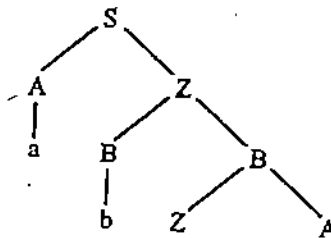


Fig. 10: Leftmost derivation tree

Case 2: Z comes from a nonterminal in s_2 .

$S \rightarrow AA$
 $A \rightarrow ZC$
 | a
 $C \rightarrow ZZ$
 $Z \rightarrow b$

Beginning of a leftmost derivation:

$S \Rightarrow AA$
 $\Rightarrow ZCA$
 $\Rightarrow bCA$
 $\Rightarrow bZZA$

Derivation tree is shown in figure 11:

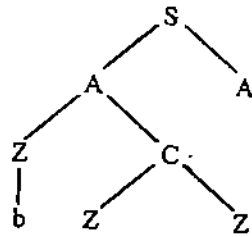


Fig. 11: Derivation Tree

In the first tree, Z is a descendant of itself. In the second, tree this is not true. Now, we will show that if a language is infinite, then we can always find an example of the first type of tree in the derivation tree of any string that is long enough.

Theorem 6: If G is a context-free grammar in Chomsky Normal Form that has p live productions, and if w is a word generated by G that has more than 2^p letters in it, then somewhere in every derivation tree for w there is an example of some non-terminal. Call it Z , being used twice where the second Z is descended from the first.

Proof: If the word w has more than 2^p letters in it, then the derivation tree for w has more than $p+1$ levels. This is because in a derivation tree drawn from a Chomsky Normal Form grammar, every internal node has either one or two children. It has one child only if that child is a leaf. At each level, there is at most twice the number of nodes as on the previous level. A leaf on the lowest level of the tree must have more than p ancestors. But, there are only p different live productions so if more than p have been used, then some live production has been used more than once. The non-terminal on the lefthand-side of this live production will appear at least twice on the path from the root to the leaf.

In a derivation, a non-terminal is said to be self-embedded if it ever occurs as a tree descendant of itself. The previous theorem says that in any context-free grammar, all sufficiently long words have leftmost derivations that include a self-embedded non-terminal. Shorter derivations may have self-embedded non-terminals, but we are guaranteed to find one in a sufficiently long derivation.

Consider the following example in which we find a self-embedded non-terminal:

$S \rightarrow AX$
 $\quad | BY$
 $\quad | AA$
 $\quad | BB$
 $\quad | a$
 $\quad | b$
 $X \rightarrow SA$
 $Y \rightarrow SB$
 $A \rightarrow a$
 $B \rightarrow b$

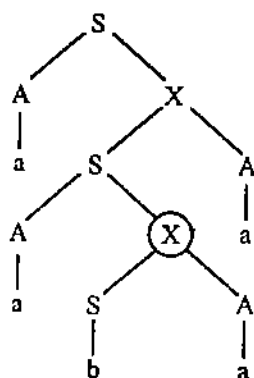


Fig. 12: A derivation Tree for the String aabaa

The production $X \rightarrow SA$ and $S \rightarrow AX$ were used twice. Let's consider the X production and think about what happens if we used this production a third time. What string would we generate? Corresponding tree is given in figure 13.

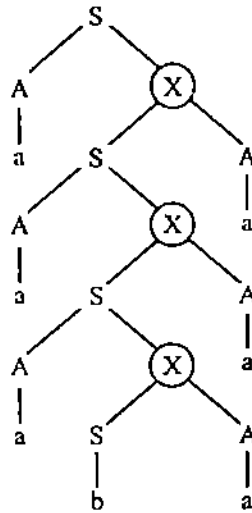


Fig. 13: Derivation Tree of String aaabaaa

This modified tree generates the string aaabaaa. We could continue reusing the rule $X \rightarrow SA$ over and over again. Can you tell what the pattern is in the strings that we would be producing?

The last use of X produces the sub-string ba. The previous X produced an a to the left of this ba and an a to the right of the ba. The X before that produced an a to the left and an a to the right. In general, X produces a^nba^n . S produces an a to the left of an X and nothing to the right. So, the strings produced by this grammar are of the form aa^nba^n . If all we wish n to signify is a, count that must be the same, then we can simplify this language description to a^nba^n for $n \geq 1$. Reusing the $X \rightarrow SA$ rule increases the number of a's in each group by one each time we use it.

Here is another example:

- $S \rightarrow AB$
- $A \rightarrow BC$
- | a
- $B \rightarrow b$
- $C \rightarrow AB$

In the derivation of the string bbabbb, $A \rightarrow BC$ is used twice. Look at the red triangular shapes in the following derivation tree. We could repeat that triangle more times and we would continue to generate words in the language.

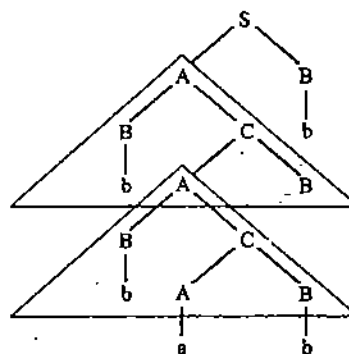


Fig. 14: Derivation tree

Pumping Lemma for Context-Free Languages

Theorem 7: If G is any context-free grammar in Chomsky Normal Form with p live productions and w is any word generated by G with length $> 2^p$, we can subdivide w into five pieces $uvxyz$ such that $x \neq \Lambda$, v and y are not both Λ and $|vxy| \leq 2^p$ and all words of the form uv^nxy^nz for $n \geq 0$ can also be generated by grammar G .

Proof: If the length of w is $> 2^p$, then there are always self-embedded non-terminals in any derivation tree for w . Choose one such self-embedded non-terminal, call it P , and let the first production used for P be $P \rightarrow QR$. Consider the part of the tree generated from the first P . This part of the tree tells us how to subdivide the string into its five parts. The sub-string vxy is made up of all the letters generated from the first occurrence of P . The sub-string x is made up of all the letters generated by the second occurrence of P . The string v contains letters generated from the first P , but to the left of the letters generated by the second P , and y contains letters generated by the first P to the right of those generated by the second P . The string u contains all letters to the left of v and the string z contains all letters to the right of y . By using the production $P \rightarrow QR$ more times, the strings v and y are repeated in place or "pumped". If we use the production $P \rightarrow QR$ only once instead of twice, the tree generates the string uxz .

Here is an example of a derivation that produces a self-embedded non-terminal and the resulting division of the string.

$S \rightarrow PQ$
 $Q \rightarrow QS$
 $\quad | \quad b$
 $p \rightarrow a$

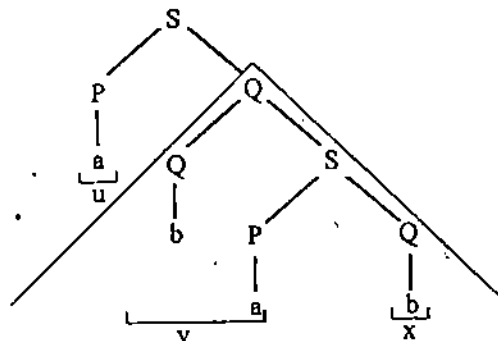


Fig. 15: A derivation tree for the string abab

Notice that the string generated by the first occurrence of Q is bab . We have a choice for which Q we take for the second one. Let's first take the one to the far right. The string generated by this occurrence of Q is b . So $x = b$ and $v = ba$. In this case, y is empty and so is z . The string $u = a$. If we pump v and y once, we get the string $a|ba|ba|b = ababab$ which is also in the language. If we pump them three times, we get $a|ba|ba|ba|ba|b = abababab$, etc.

Suppose we choose the other occurrence of Q for the second one, then we have a different sub-division of the string. In this case, the substring generated by the second

occurrence of Q is b, so $x = b$ and v is empty. The substring y , however, is ab in the case.

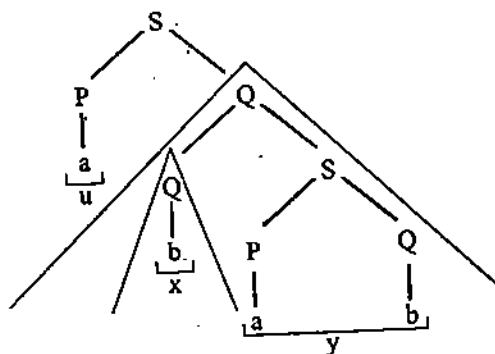


Fig. 16: Selection of u , x and y

If we pump v and y once, we get the string $a|b|ab|ab = ababab$; three times produces $a|b|ab|ab|ab|ab = abababab$, etc.

Using the Pumping Lemma for CFLs

We use the Pumping Lemma for context-free languages to prove that a language is not context-free. The proofs are always the same:

- Assume that the language in question is context-free and that the Pumping Lemma thus applies.
- Pick the string w , $|w| > 2^p$
- Sub-divide w into $uvxyz$ such that $|vxy| < 2^p$
- Pick i so that $uv^i xy^i z$ is not in the language. As in pumping lemma $uv^i xy^i z \in L$, but it is not true. So, our assumption is not correct and the language in the question is not CFL.

Here is an example:

Example 17: The language $L = \{a^n b^n a^n \mid n \geq 1\}$ is not a context-free language.

Solution: Assume that L is a context-free language. Then, any string in L with length $> 2^p$ can be sub-divided into $uvxyz$ where $uv^i xy^i z$, $n \geq 0$, are all strings in the language. Consider the string $a^{2^p} b^{2^p} a^{2^p}$ and how it might be sub-divided. Note that there is exactly one "ab" in a valid string and exactly one "ba". Neither v nor y can contain ab or ba or else pumping the string would produce more than one copy and the resulting string would be invalid. So both v and y must consist of all one kind of letters. There are three groups of letters all of which must have the same count for the string to be valid. Yet, there are only two sub-strings that get pumped, v and y . If we only pump two of the groups, we will get an invalid string.

A Stronger Version of the Pumping Lemma

There are times when a slightly stronger version of the Pumping Lemma is necessary for a particular proof. Here is the theorem:

Theorem 8: Let L be a context-free language in Chomsky Normal Form with p live productions. Then, any word w in L with length $> 2^p$ can be sub-divided into five

parts $uvxyz$ such that the length of vxy is no more than 2^p , $x \neq \Lambda$, v and y are not both Λ , and $uv^nxy^n z$, $n \geq 0$, are all in the language L .

Now, let's see a proof in which this stronger version is necessary.

Example 18: The language $L = \{a^n b^m a^n b^m \mid n, m \geq 1\}$ is not context-free.

Proof: Assume that L is a context-free language. Then, any string in L with length $> 2^p$ can be sub-divided into $uvxyz$ where $x \neq \Lambda$, v and y are not both Λ , the length of vxy is no more than 2^p , and $uv^nxy^n z$, $n \geq 0$, are all strings in the language. Consider the string $a^{2^p} b^{2^p} a^{2^p} b^{2^p}$. (The superscripts on each character are supposed to be 2^p . Some browsers can't do the double superscript.) Clearly, this string is in L and is longer than 2^p . Since the length of vxy is no more than 2^p , there is no way that we can stretch vxy across more than two groups of letters. It is not possible to have v and y both made of a 's, or v and y both made of b 's. Thus, pumping v and y will produce strings with an invalid form. Note that we need the stronger version of the Pumping Lemma because without it we can find a way to sub-divide the string so that pumping it produces good strings. We could let $u = \Lambda$, $v =$ the first group of a 's, $x =$ the first group of b 's, $y =$ the second group of a 's, and $z =$ the second group of b 's. Now, duplicating v and y produces only good strings.

Here is another example.

Example 19: $\text{DOUBLEWORD} = \{ss \mid s \in \{a,b\}^*\}$ is not a context-free language.

Proof: The same proof as we used in the last case works here. Consider the string $a^{2^p} b^{2^p} a^{2^p} b^{2^p}$ (again supposed to be double superscripts.) It is not possible to have v and y both made of the same kind of letter, so pumping will produce strings that are not in DOUBLEWORD .

Now try some exercises

Ex.8) Show that the language $\{a^{n^2} \mid n \geq 1\}$ is not context free.

Ex.9) Show that the language $\{a^p \mid p \text{ is prime}\}$ is not context free.

3.6 EQUIVALENCE OF CFG AND PDA.

In Unit 2, we established the equivalence of regular languages/expressions and finite automata. Similarly, the context-free grammar and pushdown automata, models are equivalent in the sense these define the same set of languages.

Theorem 9: Every context free Grammar is accepted by some pushdown automata.

Let $G = (V, T, R, S)$ be the given grammar where the components have the following meaning

- V : The set of variables
- T : The set of terminals
- R : The set of rules of the form
 $A \rightarrow \alpha$ with $A \in V$ and $\alpha \in (V \cup T)^*$,
i.e., α is a string of terminals and non-terminals.
- S : The start symbol.

Now, in terms of the given Grammar G , we achieve our goal through the following three parts:

- (i) We construct a PDA say $P = (Q, \Sigma, \Gamma, S, q_0, Z_0, F)$ where the components like Q, Σ etc., are expressed in terms of the known entities V, T, R, S or some other known entities.
- (ii) To show that if string $\alpha \in L(G)$, then α is accepted by the PDA constructed by (i) above.
- (iii) Conversely, if α is a string accepted by the PDA P constructed above, then $\alpha \in L(G)$

Part I: For the construction of the PDA $P = (Q, \Sigma, \Gamma, S, q_0, Z_0, F)$, we should define the values of the various components Q, Σ , etc., in terms of already known components V, T, R, S of the grammar G or some other known or newly introduced symbols.

We define the components of P as follows:

- (i) Q = the set of states of PDA = $\{q\}$, q is the only state of Q , and q is some new symbol not involved in V, T, R and S
- (ii) Σ = the set of tape symbols of P , the proposed PDA
= T (the terminals of the given grammar G)
- (iii) Γ = the stack symbols of P , = $(T \cup V)$
= the set of all symbols which are terminal or non-terminals in the given grammar G
- (iv) q_0 = initial state = q (the only state in Q is naturally the initial state also)
- (v) $Z_0 = S$, the start symbol of the given grammar G
- (vi) $F = \{q\}$,
 q being the only state in the PDA P , is naturally the only final state of the PDA.
- (vii) Next, we show below how the required function
 $\delta: Q \times \Sigma \times \Gamma \rightarrow \text{Power Set of } (Q \times \Gamma)$
is obtained in terms of the known entities Q, V, T, R and S .

- (a) For each rule $A \rightarrow \beta$ in R of the grammar G with $A \in V$ and $\beta \in (V \cup T)^*$, we define

$$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \rightarrow \beta \text{ is a rule in } R\}$$

(Note: There may be more than one rules with the some L.H.S, for example $A \rightarrow \beta a$ and $A \rightarrow b B C D$)

- (b) each (terminal) $a \in T$,
 $\delta(q, a, a) = \{(q, \epsilon)\}$

This completes the definition of P , the required PDA. Next our job is to show that $L(G)$, the language generated by G is same as $N(P)$, the language accepted by P , the PDA which we have designed above.

Proof of Parts II and III are based on the proof of the following:

Lemma Let $S = Y_0 \rightarrow Y_1 \dots \rightarrow Y_n = w = a_1 a_2 \dots a_n \in L(G)$
be a left-most derivation of w from grammar G ,
where

$$Y_i \rightarrow Y_{i+1}$$

is obtained by single application of left-most derivation, using some rule of R of the grammar G ,

Then, to each Y_i , there is a unique configuration / ID of the PDA as explained below so that Y_n corresponds to the configuration of PDA which accepts w :

Let

$$Y_i = x_i \alpha_i$$

where $x_i \in T^*$ and $\alpha_i \in (V \cup T)^*$.

Then

the string Y_i of the derivation

$$Y_0 = S \Rightarrow Y_1 \Rightarrow Y_2 \dots \Rightarrow Y_i \dots \Rightarrow Y_n = w$$

is made to correspond to the ID (y_i, α_i) of the pushdown automata constructed in Part I. The correspondence is diagrammatically shown in figure 17 where y_i is the yet-to-be scanned part of the string w on the tape, and the first terminal in Y_i is being scanned by the Head Tape:

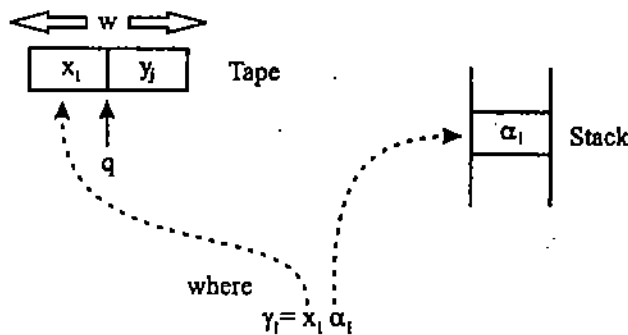


Fig.17

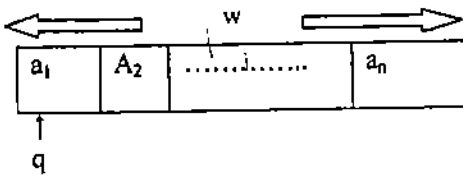
Proof of the Lemma

We prove the lemma by induction on i of Y_i

Base Case: $i = 0$

$$Y_0 = S$$

and initially the Head scans the left-most symbol on the tape, i.e.,

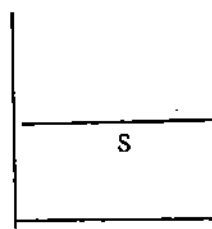


thus

$$Y_0 = x_0 S$$

where $x_0 = \epsilon$ = empty string

$$S \in \Gamma^*$$



Induction hypothesis:

We assume that if $Y_j = x_j \alpha_j$ for $j = 1, 2, \dots, i$. (where each α_j starts with a non-terminal) for each of Y_0, Y_1, \dots, Y_i , the correspondence between j^{th} strings Y_j in the derivation of w from S and the configuration config (j) given in figure 18

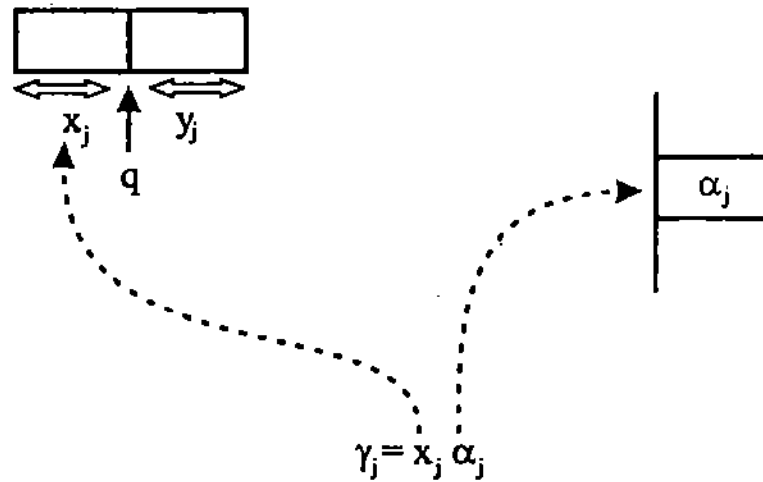


Fig. 18

Induction step

To show that the correspondence is preserved for $j = i + 1$ also. There is no loss of generality if we assume that if $\alpha_j \neq \epsilon$ then

$$\alpha_j = D_j \beta_j \quad \text{for } j = 1, \dots, i$$

where D_j is a non-terminal symbol in the grammar.

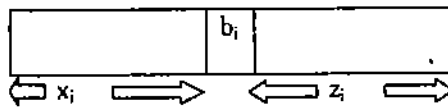
Let b_i be the first symbol of y_i (where b_i is one of the a_j 's)

$$\text{i.e. } y_i = b_i z_i$$

where z_i is a string of terminals.

$$Y_{i+1} = x_i D_{i+1} \xi_{i+1} \alpha_j$$

As



$w \in L(G)$

$$D_i \rightarrow a_{i1} \dots a_{ik} D_{i+1} \dots$$

$$\therefore x_i a_{i1} \dots a_{ik} D_{i+1} \dots$$

$x_i a_{i1}$ must be a prefix of at least
then there must be either a production

$$D_i \rightarrow b_i \dots D_{i+1} \dots$$

a production in G

or A sequence of production

$$D_i \rightarrow D_{i1} \dots$$

$$D_{i1} \rightarrow D_{i2} \dots$$

$$D_{ik} \rightarrow b_i \dots$$

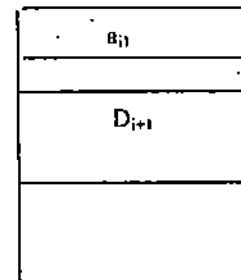
Without loss of generality, we assume that

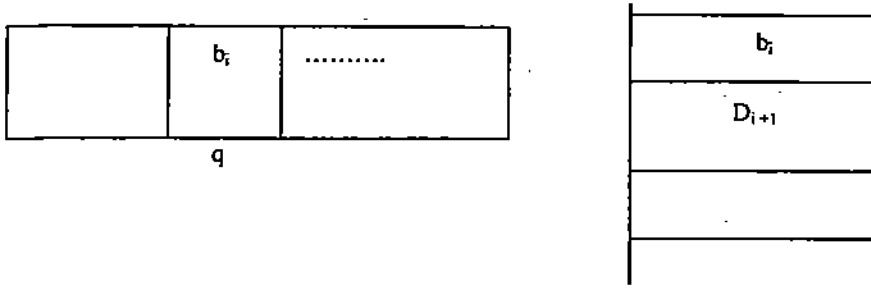
$D_i \rightarrow b_i \dots D_{i+1} \dots$, with D_{i+1} being the first non-terminal from left in the production used in Y_{i+1} from Y_i

But corresponding to this production there is a move

$$S(q, \epsilon, D_i) = (q, b_i \dots D_{i+1} \dots)$$

using this move the config becomes

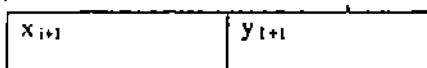




Then all the b's are popped off from the stack and Head of the tape moves to the right of the symbol next to be on the tape by the moves of the type

$$\delta(q, b_i, b) = (q, \epsilon)$$

Finally D_{i+1} is the top configuration after the execution of the above moves gives is of the form

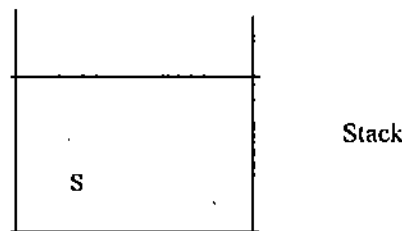
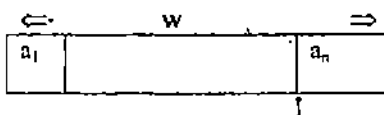


Where

α_{i+1} has a non-terminal as its left most symbol.

This completes the proof of the lemma.

Next, the lemma establishes a one-to-one correspondence between the strings Y_i in the derivations of w in the grammar G and the configurations of the pushdown automata constructed in Part I, in such a manner that $Y_n = w$ correspond to the following configuration that indicates acceptance of w and vice-versa.



This completes the proof of the Part II and Part III.

3.7 SUMMARY

In this unit we have considered the recognition problem and found out whether we can solve it for a larger class of languages. The corresponding acceptor for the context-free languages are PDA's. There are some languages which are not context free. We

18-7

can prove the non-context free languages by using the pumping lemma. Also in this unit we discussed about the equivalence two approaches, of getting a context free language. One approach is, using context free grammar and other is Pushdown Automata.

3.8 SOLUTIONS/ANSWERS

Ex.1) (a) $S \rightarrow S[S] \rightarrow [S] \rightarrow []$.

(b) $S \rightarrow S[S] \rightarrow [S] \rightarrow [S[S]] \rightarrow [[S] \rightarrow [[]]$.

Similarly rest part can be done.

Ex.2) (a) $S \rightarrow aSb/aAb$

$A \rightarrow bA/b$

Ex.3) (a) $S \rightarrow aSa/bSb/\wedge$

(b) $S \rightarrow aSa/bSb/a/b$.

Ex.4) (a) $S \rightarrow ASB$ (type 2 production)

$S \rightarrow b$ (type 3 production)

$A \rightarrow aA$ (type 3 production)

So the grammar is of type 2.

(b) $S \rightarrow aSa$ (type 2 production)

$S \rightarrow bSb$ (type 2 production)

$S \rightarrow a$ (type 3 production)

$S \rightarrow b$ (type 3 production)

$S \rightarrow \wedge$ (type 3 production)

So the grammar is of type 2.

(c) Type 2.

Ex.5) (a) $S \rightarrow AB$

$S \rightarrow aAb^5/\wedge$

$B \rightarrow b^2Ba/\wedge$

(b) $S \rightarrow AB$

$A \rightarrow a . b^2/\wedge$

$B \rightarrow b^4Ba/\wedge$

Ex.6) Suppose language is $\{wcw^T : w \in \{a,b\}^*\}$ then pda is

$(0, a, x, \text{push}(a), 0), (0, b, x, \text{push}(b), 0),$

$(0, a, a, \text{push}(a), 0), (0, b, a, \text{push}(b), 0),$

$(0, a, b, \text{push}(a), 0), (0, b, b, \text{push}(b), 0),$

$(0, c, a, \text{nop}, 1), (0, c, b, \text{nop}, 1),$

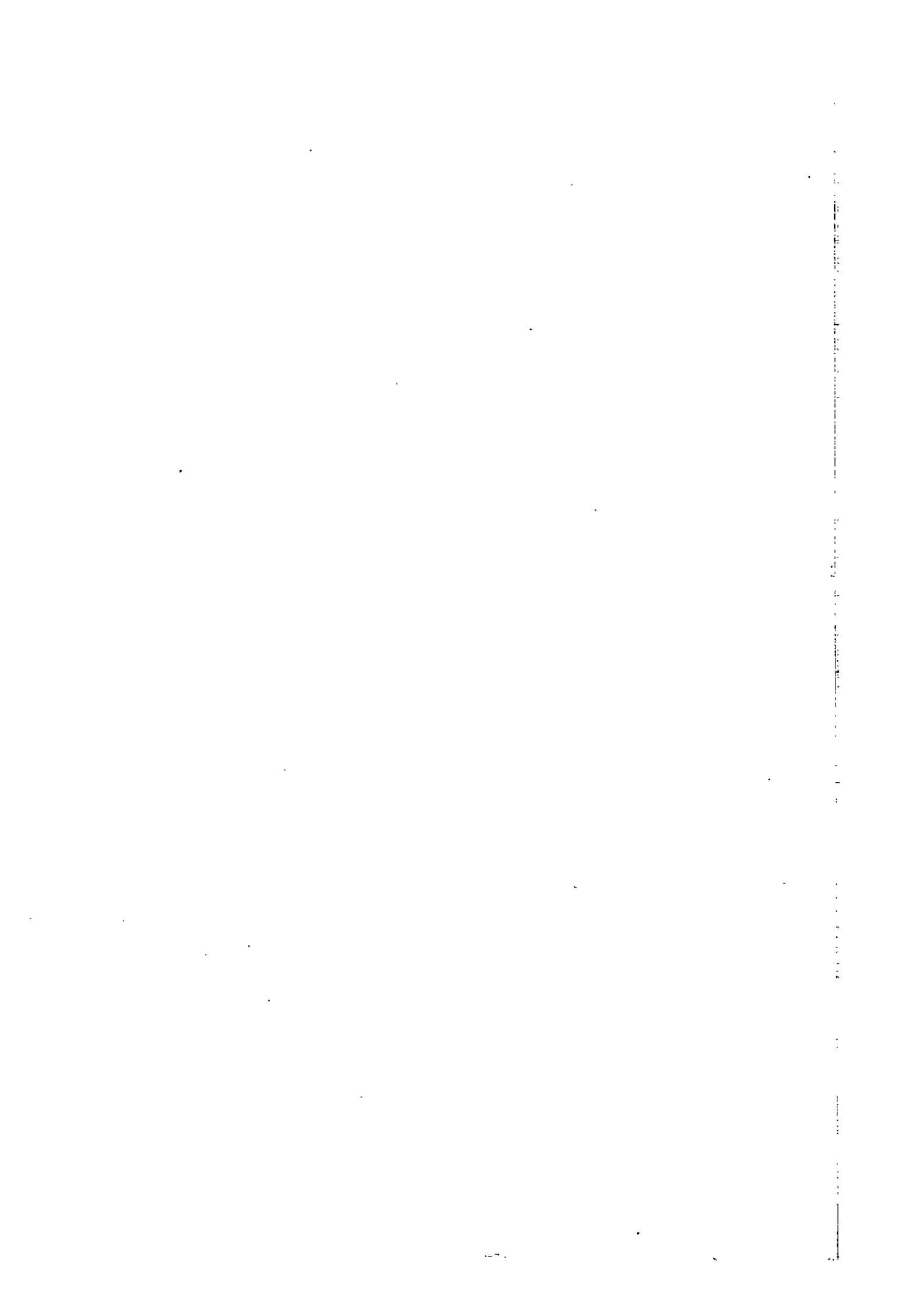
$(0, c, x, \text{nop}, 1), (1, a, a, \text{pop}, 1),$

$(1, b, b, \text{pop}, 1), (1, \wedge, x, \text{nop}, 2),$

Ex.7) Language is $\{ww^i : w \in \{a,b\}^*\}$. Similarly as Ex 6.

Ex.8) Apply pumping lemma to get a contradiction. The proof is similar to the proof that the given language is not regular.

Ex.9) Apply pumping lemma to get a contradiction. The proof is similar to the proof that the given language is not regular.





Uttar Pradesh
Rajarshi Tandon Open University

BCA-18
THEORY OF
COMPUTATION

Block

2

TURING MACHINE AND RECURSIVE FUNCTIONS

UNIT 1

Turing Machine 5

UNIT 2

Turing Machine — Miscellany 55

UNIT 3

Recursive Function Theory 92

BLOCK INTRODUCTION

"In truth, it is not knowledge, but learning; not possessing, but production; not being there, but traveling there, which provides the greatest pleasure. When I have completely understood something, then I turn away and move on into the dark; indeed, so curious is the insatiable man, that when he has completed one house, rather than living in it peacefully, he starts to build another."

Letter from C.F. Gauss to W. Bolyai on Sept. 2, 1808

Continuing in Gauss' vein, after completing the houses (i.e, models) of Finite Automata, Pushdown Automata, Regular Grammars, and Context-Free Grammars, let us build those of Turing Machines, Context-Sensitive Grammars, and Partial Recursive Functions to have better insight* in the phenomena of computations.

After having studied in previous blocks, Finite Automata and Pushdown Automata models of machine-based approach and Regular Grammar and Context-Free Grammar models of grammatical approach, both approaches for theoretical studies of the notion of *computation*, and hence, for theoretical study of problem solving using computer as a tool; in this block we extend the respective models to get still more powerful model of computation for each of the two approaches. **Turing Machine** is the next model of computation under machine-based approach and **Context-Sensitive or Phrase-Structure Grammar** is the corresponding model under grammatical approach. It may be mentioned that each of the TM model and the corresponding phrase-structure grammar model, is not just another model of computation. But, going by the current knowledge of the discipline of the Theory of Computation, each appears to be the **ultimate model** of computation.

In the first two units of this block, we study various issues related to TMs, viz, formal definitions, TM as a computer of functions, extensions of TM and their equivalences, how a TM solves a problem etc.

In Unit 3, we introduce a new approach to computation viz, Recursive Function Theory. Again, we consider various models, viz primitive recursive function, μ -recursive function and partial recursive function models in this approach to computation.

In the next and final block, viz, Block 3 of the course, we study some applications of the concepts developed in the first two blocks. Also, we study unsolvability of many problems by computational means and complexity of those problems, which are solvable.

* The purpose of computation is insight, not symbol processing, as per R.W. Hamming's: "The purpose of numerical analysis is insight, not numbers".

.....

UNIT 1 TURING MACHINE*

Structure	Page Nos.
1.0 Introduction	5
1.1 Objectives	7
1.2 Prelude to Formal Definition	7
1.3 Turing Machine: Formal Definition and Examples	9
1.4 Instantaneous Description and Transition Diagram	14
1.5 Some Formal Definitions	17
1.6 Observations	20
1.7 Turing Machine as Computer of Functions	22
1.8 Modular Construction of Complex Turing Machines	32
1.9 Summary	44
1.10 Solutions/Answers	45
1.11 Further Readings	54

1.0 INTRODUCTION

Every system—natural or man-made, must be continuously, involved in some form of **computation** in its attempt at preserving its identity as a system.

The earth, revolves around the Sun along almost identical paths, revolution after revolution; being almost at the corresponding points in the paths after a specific period of time within the revolutions. So is true of every planet in the solar system. To be at the corresponding points in their paths, revolution after revolution, must involve some computation within the solar system. But, the same should be true of any system, not just of the solar system. Thus, phenomenon of **computation** is as universal as is the phenomena of **motion**. In order to have better understanding of the phenomena of motion, we think of different approaches, use some models and formulate some principles.

Similarly, attempts at *capturing the essence of the universal phenomenon of computation* are made through various approaches, models and principles.

As happens in the case of modeling of *motion*, inadequacy of one model (e.g. *Newtonian model*) in capturing essence of motions leads to another, more robust model (e.g. *Einstein's model*), so happens in the case of modeling of computation, as is discussed below.

In the previous units, we discussed two of the major *approaches* to modeling of computation viz. the automata/machine approach and linguistic/grammatical approach. Under grammatical approach, we discussed two models viz Regular Languages and Context-free Languages.

Under automata approach, we discussed two *models* viz. Finite Automata and Pushdown Automata.

Further, we defined the concept of **computational equivalence** and established that

* Turing Machine is named so, in Honour of its inventor **Alan Mathison Turing (1921-1954)**. A.M. Turing, a British, was one of the greatest scholars of the twentieth century, and made profound contributions to the foundations of computer science. On the lines of Nobel prize, in memory of Alfred B. Nobel, for some scientific disciplines; ACM, to commemorate A.M. Turing, presents since 1966 annually Turing Award to an individual for contributions of a technical nature that are judged to be of lasting and major importance to the field of computing science.

Not every problem can be solved through computational means
Gödel (1931)

If a problem can be solved by some computational means, then there is a Turing Machine that solves the problem.....Turing Machine is an ultimate model of computation
Church-Turing Thesis (1936)

- (i) **Finite Automata** computational model is computationally equivalent to Regular Language Model.
- (ii) **Push-down Automata Model** is computationally equivalent to context-free language model.
- (iii) Pushdown Automata (or equivalently Context-Free Language) model is **more powerful** computational model in **comparison** to Finite Automata (or equivalently Regular Language) model *in the sense that every language accepted by Finite Automata is also recognized by Pushdown Automata.* However, there are languages, viz the language $\{x^n y^n : n \in \mathbb{N}\}$, which are recognized by pushdown automata but not by Finite Automata.
- (iv) There are languages, including the language $\{x^n y^n z^n : n \in \mathbb{N}\}$, which are **not accepted even by Push-down automata.**

This prompts us to discuss other, still more powerful, automata models and corresponding grammar models of computation.

Turing machine (TM) is the next more powerful model of *automata approach* which recognizes more languages than Pushdown automata models do. Also **Phrase-structure model** is the corresponding grammatical model that matches Turing machines in computational power.

In this unit, we attempt a facile and smooth introduction to the concept of Turing Machine in the following order:

- *We give a formal definition of the concept and then illustrate the involved ideas through a number of examples and remarks.*
- *We show how to realize some mathematical functions as TMs.*
- *Further, we discuss how to construct more and more complex TMs through the earlier constructed TMs, starting with actual constructions (mathematically) of some simple TMs.*

In a later unit, we discuss other issues like extensions, (formal) languages, properties and equivalences, in context of TMs.

Key words: Turing Machine (TM), Deterministic Turing Machine, Non-Deterministic Turing Machine, Turing Thesis, Computation, Computational Equivalence, Configuration of TM, Turing-Acceptable Language, Turing Decidable Language, Recursively Enumerable Language, Turing Computable Function

Notations

TM	:	Turing Machine
Γ	:	Set of tape symbols, includes #, the blank symbol
Σ	:	Set of input/machine symbols, does not include #
Q	:	the finite set of states of TM
F	:	Set of final states
a, b, c, ...	:	Members of Σ
σ	:	Variable for members of Σ
\bar{x} or x	:	Any symbol of Σ other than x
#	:	The blank symbol
α, β, γ	:	Variables for String over Σ
L	:	Move the Head to the Left
R	:	Move the Head to the Right
q	:	A state of TM, i.e, $q \in Q$
s or q_0	:	The start/initial state

Exploring Randomness By Gregory J. Chaitin. Springer-Verlag (2001)

Halt or h : The halt state. The same symbol h is used for the purpose of denoting halt state for all halt state versions of TM. And then h is not used for other purposes.
 ϵ or ϵ : The empty string

$C_1 \vdash_M C_2$: Configuration C_2 is obtained from configuration C_1 in *one* move Of the machine M
 $C_1 \vdash^* C_2$: Configuration C_2 is obtained from configuration C_1 in *finite* number of moves.
 $w_1 \underline{a} w_2$: The symbol a is the symbol currently being scanned by the Head

Or

$w_1 \underline{a} w_2$: The symbol a is the symbol currently being scanned by the Head
 \uparrow

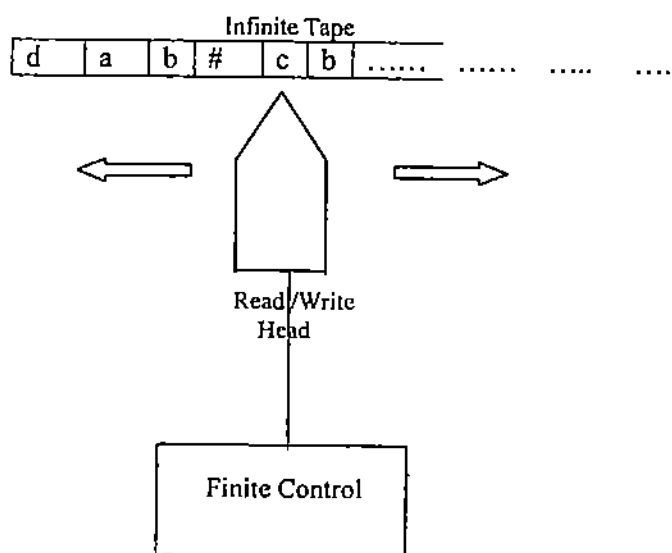
1.1 OBJECTIVES

After going through this unit, you should be able to:

- define and explain various terms mentioned under the title *key words* in the previous section.
- construct TMs for simple computational tasks
- realize some simple mathematical functions as TMs
- apply modular techniques for the construction of TMs for more complex functions and computational tasks from TMs already constructed for simple functions and tasks

1.2 PRELUDE TO FORMAL DEFINITION

In the next section, we will notice through a formal *definition of TM* that a TM is an *abstract* entity constituted of *mathematical objects* like sets and a (partial) function. However, in order to help our understanding of the subject-matter of TMs, we can *visualize* a TM as a physical computing device that can be represented as a *diagram* as shown in 1.2.1 below.



TURING MACHINE
 Fig. 1.2.1

Such a view, in addition to being more comprehensible to human beings, can be a quite *useful aid* in the design of TMs accomplishing some computable tasks, by allowing *informal* explanation of the various steps involved in arriving at a particular design.¹ Without physical view and informal explanations, whole design process would be just a sequence of derivations of new formal symbolic expressions from earlier known or derived symbolic expressions — not natural for human understanding.

According to this view of TM, it consists of

- (i) **a tape**, with an end on the left but infinite on the right side. The tape is divided into squares or *cells*, with each cell capable of holding one of the tape symbols including the blank symbol #. At any time, there can be only *finitely* many cells of the tape that can contain non-blank symbols. The set of **tape symbols** is denoted by Γ

As the very first step in the sequence of operations of a TM, the **input**, as a finite sequence of the input symbols is placed in the left-most cells of the tape. The set of input symbols denoted by Σ , does not contain the blank symbol #. However, during operations of a TM, a cell may contain a *tape symbol* which is not necessarily an input symbol.

There are versions of TM, to be discussed later, in which the tape may be infinite in both left and right sides — having neither left end nor right end.

- (ii) **a finite control**, which can be in any one of the finite number of states. *The states in TM can be divided in three categories viz.*

- (a) **the Initial state**, the state of the control just at the time when TM starts its operations. The initial state of a TM is generally denoted by q_0 or s .
(b) **the Halt state**, which is the state in which TM stops all further operations. The halt state is generally denoted by h . The halt state is distinct from the initial state. Thus, a TM HAS AT LEAST TWO STATES.
(c) **Other states**

- (iii) **a tape head (or simply Head)**, is always stationed at one of the tape cells and provides communication for interaction between the tape and the finite control. The Head can *read or scan* the symbol in the cell under it. The symbol is communicated to the finite control. The control taking into consideration the symbol and its current state decides for further course of action including

- the change of the symbol in the cell being scanned and/or
- change of its state and/or
- moving the head to the Left or to the Right. The control may decide not to move the head.

The course of action is called a move of the Turing Machine. In other words, the move is a function of current state of the control and the tape symbol being scanned.

In case the control decides for change of the symbol in the cell being scanned, then the change is carried out by the head. This change of symbol in the cell being scanned is called *writing of the cell by the head*.

Initially, the head scans the left-most cell of the tape.

Now, we are ready to consider a formal definition of a Turing Machine in the next section.

1.3 TURING MACHINE: FORMAL DEFINITION AND EXAMPLES

There are a number of versions of a TM. We consider below *Halt State version* of formal definition a TM.

Definition: Turing Machine (Halt State Version)

A Turing Machine is a sextuple of the form $(Q, \Sigma, \Gamma, \delta, q_0, h)$, where

- (i) Q is the finite set of states,
- (ii) Σ is the finite set of non-blank information symbols,
- (iii) Γ is the set of tape symbols, including the blank symbol #
- (iv) δ is the *next-move partial function* from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R, N\}$, where 'L' denotes the tape Head moves to the left adjacent cell, 'R' denotes tape Head moves to the Right adjacent cell and 'N' denotes Head does not move, i.e., continues scanning the same cell.

In other words, for $q_i \in Q$ and $a_k \in \Gamma$, there exists (not necessarily always, because δ is a partial function) some $q_j \in Q$ and some $a_l \in \Gamma$ such that $\delta(q_i, a_k) = (q_j, a_l, x)$, where x may assume any one of the values 'L', 'R' and 'N'.

The meaning of $\delta(q_i, a_k) = (q_j, a_l, x)$ is that if q_i is the current state of the TM, and a_k is cell currently under the Head, then TM writes a_l in the cell currently under the Head, enters the state q_j and the Head moves to the right adjacent cell, if the value of x is R, Head moves to the left adjacent cell, if the value of x is L and continues scanning the same cell, if the value of x is N.

- (v) $q_0 \in Q$, is the initial/start state.
- (vi) $h \in Q$ is the 'Halt State', in which the machine stops any further activity.

Remark 1.3.1

Again, there are a number of variations in literature of even the above version of TM. For example, some authors allow at one time only one of the two actions viz. (i) writing of the current cell and (ii) movement of the Head to the left or to the right. However, this restricted version of TM can easily be seen to be computationally equivalent to the definition of TM given above, because one move of the TM given by the definition can be replaced by at most two moves of the TM introduced in the Remark.

In the next unit, we will discuss different versions of TM and issues relating to equivalences of these versions.

In order to illustrate the ideas involved, let us consider the following simple examples.

Example 1.3.2:

Consider the Turing Machine $(Q, \Sigma, \Gamma, \delta, q_0, h)$ defined below that erases all the non-blank symbols on the tape, where the sequence of non-blank symbols does not contain any blank symbol # in-between:

$$Q = \{q_0, h\} \quad \Sigma = \{a, b\}, \quad \Gamma = \{a, b, \#\}$$

and the next-move function δ is defined by the following table:

q: State	σ : Input Symbol	$\delta (q, \sigma)$
q_0	a	$\{q_0, \#, R\}$
q_0	b	$\{q_0, \#, R\}$
q_0	#	$\{h, \#, N\}$
h	#	ACCEPT

Next, we consider how to design a Turing Machine to accomplish some computational task through the following example. For this purpose, we need the definition.

A string Accepted by a TM

A string α over Σ is said to be accepted by a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ if when the string α is placed in the left-most cells on the tape of M and TM is started in the initial state q_0 then after a finite number of moves of the TM as determined by δ , Turing Machine is in state h (and hence stops an further operations. The concepts will be treated in more details later on. Further, a string is said to be rejected if under the conditions mentioned above, the TM enters a state $q \neq h$ and scans some symbol x , then $\delta (q, x)$ is not defined.

Example 1.3.3

Design a TM which accepts all strings of the form $b^n d^m$ for $n \geq 1$ and rejects all other strings.

Let the TM M to be designed is given by $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ with $\Sigma = \{b, d\}$. The values of Q, Γ, δ , shall be determined by the design process explained below. However to begin with we take $\Gamma = \{b, d, \#\}$.

We illustrate the design process by considering various types of strings which are to be accepted or rejected by the TM .

As input, we consider only those strings which are over $\{b, d\}$. Also, it is assumed that, when moving from left, occurrence of first # indicates termination of strings over Γ

Case I: When the given string is of the form $b^n d^m (b | d)^k$ for $n \geq 1, m \geq 1$ as shown below for $n = 2, m = 1$

We are considering this particular type of strings, because, by taking simpler cases of the type, we can determine some initial moves of the required TM both for strings to be accepted and strings to be rejected.

b	b	d	-	-	-	-
---	---	---	---	---	---	---

Where '-' denotes one of b, d or $\#$

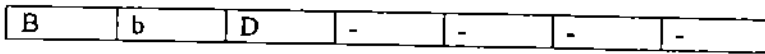
Initially, TM should mark left-most b . The term mark is used here in this sense that the symbol is scanned matching with corresponding b or d as the case may be. To begin with, the TM should attempt to match, from the left, the first b to the d which is the first d after all b 's have exhausted. For this purpose, TM should move right skipping over all b 's. And after scanning the corresponding d , it should move left, until we reach the b , which is the last b that was marked.

Next, TM should mark the b , if it exists, which is immediately on the right of the previously marked b . i.e., should mark the b which is the left-most b which is yet to be marked.

But, in order to recognize the yet-to-be-marked left-most b, we must change each of the b's, immediately on marking, to some other symbol say B. Also, for each b, we attempt to find the left-most yet-to-be-marked d. In order to identify the left-most yet-to-be-marked d, we should change each of the d's immediately on marking it, by some other symbol say D.

Thus we require two additional tape symbols B and D, i.e., $\Gamma = \{b, d, B, D, \#\}$.

After one iteration of replacing one b by B and one d by D the tape would be of the form



and the tape Head would be scanning left-most b.

In respect of the states of the machine, we observe that in the beginning, in the initial state q_0 , the cell under the Head is a b, and then this b is replaced by a B; and at this stage, if we do not change the state then TM would attempt to change next b also to B without matching the previous b to the corresponding d. But in order to recognize the form $b^n d^n$ of the string we do not want, other b's to be changed to B's before we have marked the corresponding d. Therefore

$$\delta(q_0, b) = (q_1, B, R)$$

Therefore, the state must be changed to some new state say q_1 . Also in order to locate corresponding d, the movement of the tape Head must be to the right. Also, in state q_1 , the TM Head should skip over all b's to move to the right to find out the first d from left. Therefore, even on encountering b, we may still continue in state q_1 . Therefore, we should have

$$\delta(q_1, b) = (q_1, b, R)$$

However, on encountering a d, the behaviour of the machine would be different, i.e., now TM would change the first d from left to D and start leftward journey. Therefore, after a d is changed to D, the state should be changed to say q_2 . In state q_2 we start leftward journey jumping over D's and b's. Therefore

$$\begin{aligned} \delta(q_1, d) &= (q_2, D, L) \text{ and} \\ \delta(q_2, D) &= (q_2, D, L) \text{ and} \\ \delta(q_2, b) &= (q_2, b, L) \end{aligned}$$

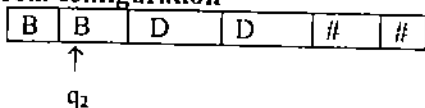
In q_2 , when we meet the first B, we know that none of the cells to the left of the current cell contains b and, if there is some b still left on the tape, then it is in the cell just to the right of the current cell. Therefore, we should move to the right and then if it is a b, it is the left-most b on the tape and therefore the whole process should be repeated, starting in state q_0 again.

Therefore, before entering b from the left side, TM should enter the initial state q_0 . Therefore

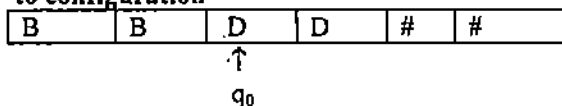
$$\delta(q_2, B) = (q_0, B, R)$$

For to-be-accepted type string, when all the b's are converted to B's and when the last d is converted to D in q_2 , we move towards left to first B and then move to right in q_0 then we get the following transition:

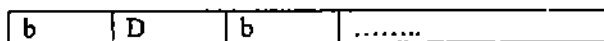
from configuration



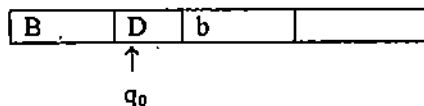
to configuration



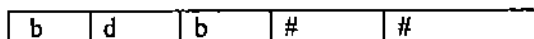
Now we consider a special subcase of $b^n d^m (b | d)^*$, in which initially we have the following input



Which after some moves changes to



The above string is to be rejected. But if we take $\delta(q_0, D)$ as q_0 then whole process of matching b's and d's will be again repeated and then even the (initial) input of the form



will be incorrectly accepted. In general, in state q_0 , we encounter D, if all b's have already been converted to B's and corresponding d's to D's. Therefore, the next state of $\delta(q_0, D)$ cannot be q_0 .

Let

$$\delta(q_0, D) = (q_3, D, R)$$

As explained just above, for a string of the to-be-accepted type, i.e., of the form $b^n d^n$, in q_3 we do not expect symbols b, B or even another d because then there will be more d's than b's in the string, which should be rejected.

In all these cases, strings are to be rejected. One of the ways of rejecting a string say s by a TM is first giving the string as (initial) input to the TM and then by not providing a value of δ in some state $q \neq h$, while making some move of the TM.

Thus the TM, not finding next move, stops in a state $q \neq h$. Therefore, the string is rejected by the TM.

Thus, each of $\delta(q_3, b)$, $\delta(q_3, B)$ and $\delta(q_3, D)$ is undefined

Further, in q_3 , we skip over D's, therefore

$$\delta(q_3, D) = (q_3, D, R)$$

Finally when in q_3 , if we meet #, this should lead to accepting of the string of the form $b^n d^n$, i.e, we should enter the state h . Thus

$$\delta(q_3, \#) = (h, \#, N)$$

Next, we consider the cases *not* covered by $b^n d^m (b | d)^*$ with $n \geq 1, m \geq 1$ are. Such

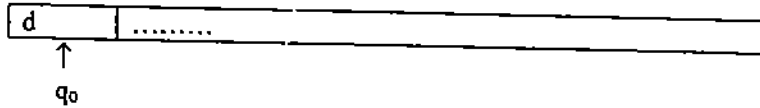
Case II when $n = 0$ but $m \neq 0$, i.e. when input string is of the form $d^m (b | d)^*$ for $m \neq 0$.

Case III when the input is of the form $b^n \#, n \neq 0$

Case IV when the input is of the form $\# \dots \dots \dots$

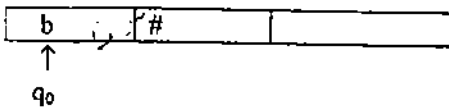
Now we consider the cases in detail.

Case II:

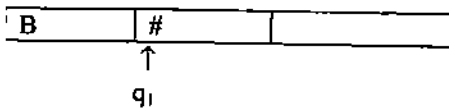


The above string is to be rejected, therefore, we take $\delta(q_0, d)$ as undefined

Case III. When the input is of the form $b^n \# \dots$, say



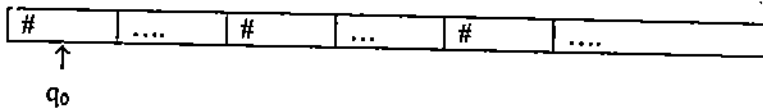
After one round we have



As the string is to be rejected, therefore,

$\delta(q_1, \#)$ is undefined

Case IV: When # is the left-most symbol in the input



As the string is to be rejected, therefore, we take $\delta(q_0, \#)$ as undefined

We have considered all possible cases of input strings over $\Sigma = \{b, d\}$ and in which, while scanning from left, occurrence of the first # indicates termination of strings over Γ .

After the above discussion, the design of the TM that accepts strings of the form $b^n d^n$ and rejects all other strings over $\{b, d\}$, may be summarized as follows:

The TM is given by $(Q, \Sigma, \Gamma, \delta, q_0, h)$ where

$Q = \{q_0, q_1, q_2, q_3, h\}$

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

$\Gamma = \{b, d, B, D, \#\}$

The next-move partial function δ is given by

	b	d	B	D	#
q_0	$\{q_1, B, R\}$	*	*	$\{q_1, D, R\}$	*
q_1	$\{q_1, b, R\}$	$\{q_2, D, L\}$	*	$\{q_1, D, R\}$	*
q_2	$\{q_2, b, L\}$	*	$\{q_0, B, R\}$	$\{q_2, D, L\}$	*
q_3	*	*	*	$\{q_3, D, R\}$	$\{h, \#, N\}$
h	*	*	*	*	Accept

* indicates the move is not defined.

Remark 1.3.4

In general, such lengthy textual explanation as provided in the above case of design of a TM, is not given. We have included such lengthy explanation, as the

purpose is to explain the very process of design. In general, table of the type given above along with some supporting textual statements are sufficient as solutions to such problems. In stead of tables, we may give Transition Diagrams (to be defined).

Ex. 1) Design a TM that recognizes the language of all strings of even lengths over the alphabet {a, b}.

Ex. 2) Design a TM that accepts the language of all strings which contain aba as a sub-string.

1.4 INSTANTANEOUS DESCRIPTION AND TRANSITION DIAGRAMS

1.4.1 Instantaneous Description

The following differences in the roles of tape and tape Head of Finite Automaton (FA) and pushdown Automaton (PDA) on one hand and in the roles of tape and tape head of Turing Machine on other hand need to be noticed:

- (i) The cells of the tape of an FA or a PDA are only read/scanned but are never changed/written into, whereas the cells of the tape of a TM may be written also.
- (ii) The tape head of an FA or a PDA always moves from left to right. However, the tape head of a TM can move in both directions.
As a consequence of facts mentioned in (i) and (ii) above, we conclude that in the case of FA and PDA the information in the tape cells already scanned do not play any role in deciding future moves of the automaton, but in the case of a TM, the information contents of all the cells, including the ones earlier scanned also play a role in deciding future moves. This leads to the slightly different definitions of configuration or Instantaneous Description (ID) in the case of a TM.

The total configuration or, for short just, configuration of a Turing Machine is the information in respect of:

- (i) Contents of all the cells of the tape, starting from the left-most cell up to atleast the last cell containing a non-blank symbol and containing all cells upto the cell being scanned.
- (ii) The cell currently being scanned by the machine and
- (iii) The state of the machine.

Some authors use the term *Instantaneous Description* instead of *Total Configuration*.

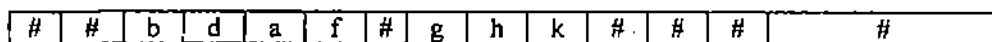
Initial Configuration: The total configuration at the start of the (Turing) Machine is called the initial configuration.

Halted Configuration: is a configuration whose state component is the Halt state

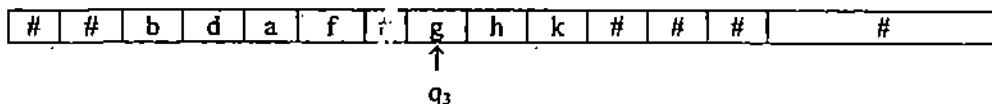
There are various *notations* used for denoting the total configuration of a Turing Machine.

Notation 1: We use the notations, illustrated below through an example:

Let the TM be in state q_3 scanning the symbol g with the symbols on the tape as follows:



Then one of the notations is



Notation 2: However, the above being a two-dimensional notation, is sometimes inconvenient. Therefore the following linear notations are frequently used: $(q_3, \# \# b d a f \# \underline{g} h k)$, in which third component of the above 4-component vector, contains the symbol being scanned by the tape head.

Alternatively, the configuration is also denoted by $(q_3, \# \# b d a f \# \underline{g} h k)$, where the symbol under the tape head is underscored but two last commas are dropped.

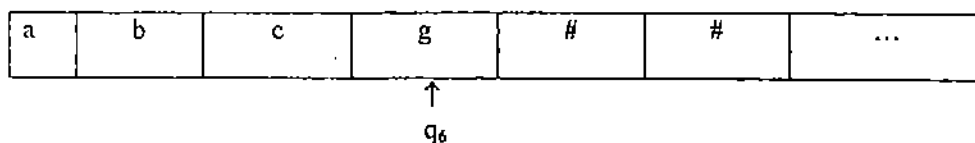
It may be noted that the sequence of blanks after the last non-blank symbol, is not shown in the configuration. The notation may be alternatively written $(q_3, w, \underline{g}, u)$ where w is the string to the left and u the string to the right respectively of the symbol that is currently being scanned.

In case g is the left-most symbol then we use the empty string ϵ instead of w . Similarly, if g is being currently scanned and there is no non-blank character to the right of g then we use ϵ , the empty string instead of u .

Notation 3: The next notation neither uses parentheses nor commas. Here the state is written just to the left of the symbol currently being scanned by the tape Head. Thus the configuration $(q_3, \# \# b d a f \# \underline{g}, h, k)$ is denoted as $\# \# b d a f \# q_3 \underline{g} h k$
Thus if the tape is like



then we may denote the corresponding configuration as $(q_5, \epsilon, \underline{g}, u)$. And, if the tape is like



Then the configuration is $(q_6, a b c, \underline{g}, \epsilon)$ or $(q_6, a b c \underline{g})$ or alternatively as $a b c q_6 \underline{g}$ by the following notation.

1.4.2 Transition Diagrams

In some situations, graphical representation of the next-move (partial) function δ of a Turing Machine may give better idea of the behaviour of a TM in comparison to the tabular representation of δ .

A **Transition Diagram** of the next-move functions δ of a TM is a graphical representation consisting of a finite number of nodes and (directed) labelled arcs between the nodes. Each node represents a state of the TM and a label on an arc from one state (say p) to a state (say q) represents the information about the required input symbol say x for the transition from p to q to take place and the action on the part of

the control of the TM. The action part consists of (i) the symbol say y to be written in the current cell and (ii) the movement of the tape Head.

Then the label of an arc is generally written as $x/(y, M)$ where M is L, R or N.

Example 1.4.2.1

Let $M = \{Q, \Sigma, \Gamma, \delta, q_0, h\}$
 Where $Q = \{q_0, q_1, q_2, h\}$
 $\Sigma = \{0, 1\}$
 $\Gamma = \{0, 1, \#\}$
 and δ be given by the following table.

	0	1	#
q_0	-	-	$(q_2, \#, R)$
q_1	$(q_2, 0, R)$	$(q_1, \#, R)$	$(h, \#, N)$
q_2	$(q_2, 0, L)$	$(q_1, 1, R)$	$(h, \#, N)$
h	-	-	-

Then, the above Turing Machine may be denoted by the Transition Diagram shown below, where we assume that q_0 is the initial state and h is a final state.

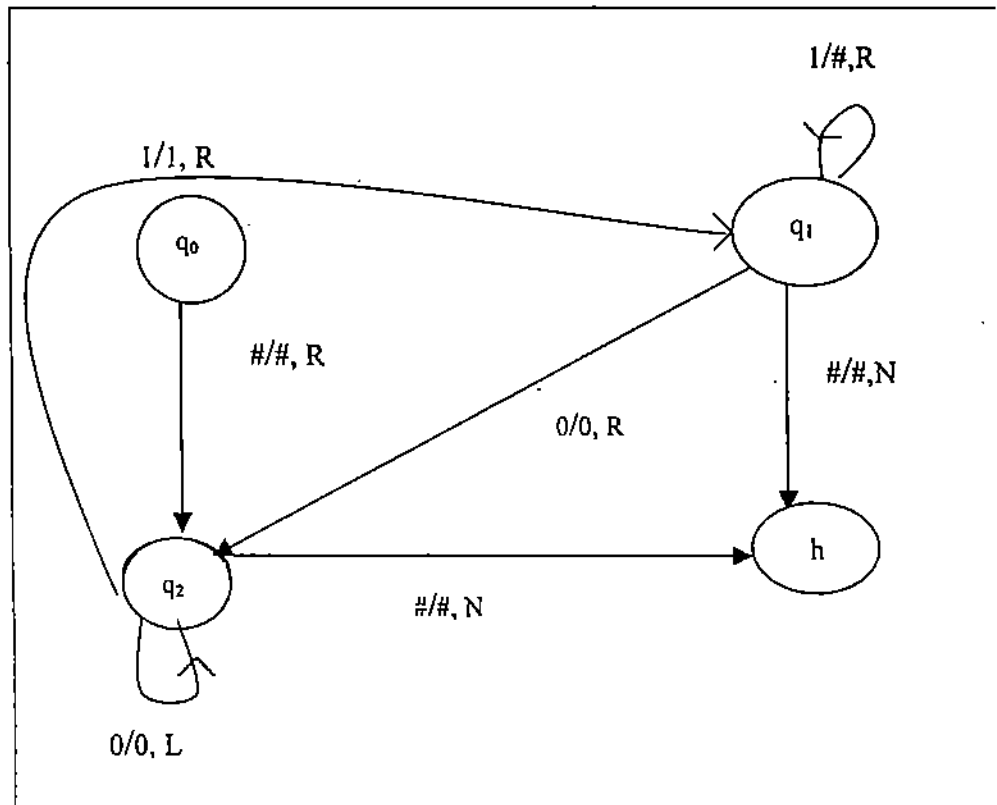


Fig. 1.4.2.1

- Ex. 3) Design a TM M that recognizes the language L of all strings over $\{a, b, c\}$ with
- (i) number of a's = Number of b's = Number of c's and
 - (ii) if (i) is satisfied, the final contents of the tape are the same as the input, i.e. the initial contents of the tape are also the final contents of the tape, else rejects the string.
- Ex. 4) Draw the Transition Diagram of the TM that recognizes strings of the form $b^n d^n$, $n \geq 1$ and was designed in the previous section.

Ex. 5) Design a TM that accepts all the language of all palindromes over the alphabet $\{a, b\}$. A palindrome is a string which equals the string obtained by reversing the order of occurrence of letters in it. Further, find computations for each of the strings (i) babb (ii) bb (iii) bab.

Ex. 6) Construct a TM that copies a given string over $\{a, b\}$. Further find a computation of the TM for the string aab.

1.5 SOME FORMAL DEFINITIONS

In the previous sections of the unit, we have used, without formally defining some of the concepts like *move*, *acceptance* and *rejection* of strings by a TM. In this section, we define these concepts formally

In the rest of the section we assume the TM under consideration is

$$M = (Q, \Sigma, \Gamma, \delta, q_0, h)$$

Definition: Move of a Turing Machine. We give formal definition of the concept by considering three possible different types of moves, viz.

- 'move to the left',
- 'move to the right', and
- 'Do not Move'.

For the definition and notation for Move, assume the TM is in the configuration $(q, a_1 a_2 \dots a_{i-1}, a_i, a_{i+1} \dots a_n)$

Case (i) $\delta(a_i, q) = \delta(b, p, L)$, for motion to left

Consider the following three subcases:

Case i(a) if $i > 1$, then the move is the activity of TM of going from the configuration

$(q, a_1 a_2 \dots a_{i-1}, a_i, a_{i+1} \dots a_n)$ to the configuration $(p, a_1 \dots a_{i-2}, a_{i-1}, a_i a_{i+1} \dots a_n)$ and is denoted as $(q, a_1 a_2 \dots a_{i-1}, a_i, a_{i+1} \dots a_n) \vdash_m (p, a_1 \dots a_{i-2}, a_{i-1}, a_i a_{i+1} \dots a_n)$.

The suffix M, denoting the TM under consideration, may be dropped, if the machine under consideration is known from the context.

Case i(b) if $i = 1$, the move leads to hanging configuration, as TM is already scanning left-most symbol and attempts to move to the left, which is not possible. Hence move is not defined.

Case i(c) when $i = n$ and b is the blank symbol $\#$, then the move is denoted as

$(q, a_1 a_2 \dots a_{n-1}, a_n, c) \vdash (q, a_1 a_2 \dots a_{n-2}, a_{n-1}, \epsilon, c)$.

Case (ii) $\delta(a_i, q) = \delta(b, p, R)$, for motion to the right

Consider the following two subcases:

Case ii(a) if $i < n$ then the move is denoted as

$(q, a_1 \dots a_{i-1}, a_i, a_{i+1} \dots a_n) \vdash (p, a_1, \dots, a_{i-1}, b, a_{i+1}, a_{i+2} \dots a_n)$

Case ii(b) if $i = n$ the move is denoted as

$(q, a_1 \dots a_{n-1}, a_n, c) \vdash (p, a_1 \dots, \#, c)$

Case (iii) $\delta(a_i, q) = (b, p, \text{'No Move'})$ when Head does not move.

then the move is denoted as
 $(q, a_1 \dots a_{i-1}, a_i, a_{i+1} \dots a_n) \vdash (p, a_1 \dots a_{i-1}, b, a_{i+1} \dots a_n)$

Definition: A configuration results (or is derived) from another configuration:

We illustrate the concept through an example based on say Case (iii) above of the definition of 'move'. In this case, we say the configuration $(p, a_1 \dots a_{i-1}, b, a_{i+1} \dots a_n)$ results in a single move or is derived in a single move from the configuration $(q, a_1 \dots a_{i-1}, a_i, a_{i+1} \dots a_n)$. Also, we may say that the move yields the configuration $(p, a_1 \dots a_{i-1}, b, a_{i+1} \dots a_n)$ or the configuration $(q, a_1 \dots a_{i-1}, a_i, a_{i+1} \dots a_n)$ yields the configuration $(p, a_1 \dots a_{i-1}, b, a_{i+1} \dots a_n)$ in a single move.

Definition: Configuration results in n Moves or finite number of moves:

If, for some positive integer n, the configurations $c_1, c_2 \dots c_n$ are such that c_i results from c_{i-1} in a single move, i.e.,

$$c_{i-1} \vdash c_i \quad \text{for } i = 2, \dots, n$$

then, we may say that c_n results from c_1 in n moves or a finite number of moves. The fact is generally denoted as

$$c_1 \vdash^n c_n \quad \text{or} \quad c_1 \vdash^* c_n$$

The latter notation is the preferred one, because generally n does not play significant role in most of the relevant discussions.

The notation $c_1 \vdash^* c_n$ also equivalently stands for the statement that c_1 yields c_n in finite number of steps.

Definition: Computation

If c_0 is an initial configuration and for some n, the configurations c_1, c_2, \dots, c_n are such that $c_0 \vdash c_1 \vdash \dots \vdash c_n$, then, the sequence of configurations $c_0, c_1 \dots c_n$ constitutes a computation

Definition: A string $\omega \in \Sigma^*$ acceptable by a TM

ω is said to be acceptable by TM M if $(q_0, \omega) \vdash^* (h, \gamma)$ for $\gamma \in \Gamma^*$

Informally, ω is acceptable by M, if when the machine M is started in the initial state q_0 after writing ω on the leftmost part of the tape, then, if after finite number of moves, the machine M halts (i.e., reaches state h and of course, does not hang and does not continue moving for ever) with some string γ of tape symbols, the original string ω is said to be accepted by the machine M

Definition: Length of computation

If C_0 is initial configuration of a TM M and C_0, C_1, \dots, C_n is a computation, then n is called the length of the computation C_0, C_1, \dots, C_n .

Definition: Input to a computation

In the initial configuration, the string, which is on that portion of the tape, beginning with the first non-blank square and ending with the last non-blank square, is called input to the computation.

Definition: Language accepted by a TM

$M = (\theta, \Sigma, \Gamma, \delta, q_0, h)$, denoted by $L(M)$, and is defined as

$L(M) = \{ \omega \mid \omega \in \Sigma^* \text{ and if } \omega = a_1 \dots a_n \text{ then}$

$$(q_0, e, a_1, a_2, \dots, a_n) \vdash^*$$

$$(h, b_1 \dots b_{j-1}, b_j, \dots, b_{j+1}, \dots, b_n)$$

for some $b_1 b_2 \dots b_n \in \Gamma^*$

$L(M)$, the language accepted by the TM M is the set of all finite strings ω over Σ which are accepted by M .

Definition: Turing Acceptable Language

A language L over some alphabet is said to be *Turing Acceptable Language*, if there exists a Turing Machine M such that $L = L(M)$

Definition: Turing Decidable Language

There are at least two alternate, but of course, equivalent ways of defining a Turing Decidable Language as given below

Definition: A language L over Σ , i.e, $L \subseteq \Sigma^*$ is said to be Turing Decidable, if both the languages L and its complement $\Sigma^* - L$ are Turing acceptable.

Definition: A language L over Σ , i.e, $L \subseteq \Sigma^*$ is said to be Turing Decidable, if there is a function

$$f_L: \Sigma^* \rightarrow \{ \underline{Y}, \underline{N} \}$$

such that for each $\omega \in \Sigma^*$

$$f_L(\omega) = \begin{cases} \underline{Y} & \text{if } \omega \in L \\ \underline{N} & \text{if } \omega \notin L \end{cases}$$

Remark 1.5.1

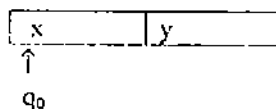
A very important fact in respect of Turing acceptability of a string (or a language) needs our attention. The fact has been discussed in details in a later unit about undecidability. However, we briefly mention it below.

For a TM M and an input string $\omega \in \Sigma^*$, even after a large number of moves we may not reach the halt state. However, from this we can neither conclude that 'Halt state will be reached in a finite number of moves' nor can we conclude that 'Halt state will not be reached in a finite number moves'.

This raises the question of how to decide that an input string w is *not* accepted by a TM M .

An input string w is said to be '*not accepted*' by a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ if any of the following three cases arise:

- (i) There is a configuration of M for which there is no next move i.e., there may be a state and a symbol under the tape head, for which δ does not have a value.
- (ii) The tape Head is scanning the left-most cell containing the symbol x and the state of M is say q and $\delta(x, q)$ suggests a move to the 'left' of the current cell. However, there is no cell to the left of the left-most cell. Therefore, move is not possible. The potentially resulting situation (can't say exactly configuration) is called **Hanging configuration**.
- (iii) The TM on the given input w enters an infinite loop. For example if configuration is as



and we are given

$\delta(q_0, x) = (q_1, x, R)$
and $\delta(q_1, y) = (q_0, y, L)$
Then we are in an infinite loop.

1.6 OBSERVATIONS

The concept of TM is one of the most important concepts in the theory of Computation. In view of its significance, we discuss a number of issues in respect of TMs through the following remarks.

Remark 1.6.1

Turing Machine is not just another computational model, which may be further extended by another still more powerful computational model. It is *not only the most powerful computational model known so far but also is conjectured to be the ultimate computational model*. In this regard, we state below the

Turing Thesis: *The power of any computational process is captured within the class of Turing Machines.*

It may be noted that Turing thesis is just a *conjecture and not a theorem, hence, Turing Thesis* can not be logically deduced from more elementary facts. *However*, the conjecture can be shown to be *false*, if a more powerful computational model is proposed that can recognize all the languages which are recognized by the TM model *and also* recognizes at least one more language that is not recognized by any TM.

In view of the unsuccessful efforts made in this direction since 1936, when Turing suggested his model, at least at present, it seems to be unlikely to have a more powerful computational model than TM Model.

Remark 1.6.2

The *Finite Automata* and *Push-Down Automata models* were used only as **accepting devices** for languages in the sense that the automata, when given an input string from a language, tells whether the string is *acceptable* or not. *The Turing Machines are designed to play at least the following three different roles:*

- (i) **As accepting devices for languages**, similar to the role played by FAs and PDAs.
- (ii) **As a computer of functions**. In this role, a TM represents a particular function (say the *SQUARE* function which gives as output the square of the integer given as input). *Initial input* is treated as representing an **argument** of the function. And the (*final*) string on the tape when the TM enters the *Halt State* is treated as representative of the **value** obtained by an application of the function to the argument represented by the initial string.
- (iii) **As an enumerator of strings of a language** that outputs the strings of a language, one at a time, in *some systematic order*, i.e. as a list.

Remark 1.6.3

Halt State of TM vs. set of Final States of FA/PDA

We have already briefly discussed the differences in the behaviour of TM on entering the Halt State and the behaviour of Finite Automata or Push Down Automata on entering a Final State.

A TM on entering the Halt State stops making moves and whatever string is there on the tape, is taken as output irrespective of whether the position of Head is at the end or in the middle of the string on the tape. However, an FA/PDA, while scanning a symbol of the input tape, if enters a final state, can still go ahead (*as it can do on entering a non-final state*) with the repeated activities of moving to the right, of scanning the symbol under the head and of entering a new state etc. In the case of FA / PDA, the portion of string from left to the symbol under tape Head is accepted if the state is a final state and is not accepted if the state is not a final state of the machine.

To be more clear we repeat: the only *difference* in the two situations when an FA/PDA enters a final state and when it enters a non-final state is that in the case of the *first situation*, the part of the input scanned so far is said to be **accepted/recognized**, whereas in the *second situation* the input scanned so far is said to be **unaccepted**.

Of course, in the Final State version of TM (discussed below), the Head is allowed movements even after entering a Final State. Some definite statement like 'Accepted/Recognized' can be made if, in this version, the TM is in Final State.

Remark 1.6.4

Final State Version of Turing Machine

Instead of the version discussed above, in which a particular state is designated as *Halt State*, some authors define TM in which a subset of the set of states Q is designated as *Set of Final States*, which may be denoted by F . This version is extension of Finite automata with the following changes, which are minimum required changes to get a Turing Machine from an FA.

- (i) The Head can move in both Left and Right directions whereas in PDA/FA the head moves only to the Right.
- (ii) The TM, while scanning a cell, can both read the cell and also, if required, change the value of the cell, i.e., can *write in the cell*. In Finite Automata, the Head *only* can read the cell. It can be shown that the *Halt State* version of TM is *equivalent* to the *Final State* version of Turing Machine.
- (iii) In this version, the TM machine halts only if in a given state and a given symbol under the head, no next move is possible. Then the (initial) input on the tape of TM, is unacceptable.

Definition: Acceptability of $w \in \Sigma^*$ in Final State Version

Let $M_1 = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a TM in final state version. Then w is said to be acceptable if C_0 is the initial configuration with w as input string to M_1 and

$$C_0 \vdash^* C_n$$

is such that

$$C_n = (p, \alpha, a, \beta)$$

with p in F , set of final states, and $a \in \Gamma$, the set of tape symbols, and $\alpha, \beta \in \Gamma^*$

Equivalence of the Two Versions

We discuss the equivalence only informally. If in the Halt state version of a TM instead of the halt state h , we take $F = \{h\}$ then it is the Final state version of the TM. Conversely, if $F = \{f_1, f_2, \dots, f_r\}$ is the set of final states then we should note the fact that in the case of acceptance of a string, a TM in final state version enters a final state *only once* and then halts with acceptance. Therefore if we rename each of the final state as h , it will not make any difference to the computation of an acceptable or

unacceptable string over Σ . Thus F may be treated as $\{h\}$, which further may be treated as just h .

1.7 TURING MACHINES AS COMPUTER OF FUNCTIONS

In the previous section of this unit, we mentioned that a Turing Machine may be used as

- (i) A language Recognizer/acceptor
- (ii) A computer of Functions
- (iii) An Enumerator of Strings of a language.

We have already discussed the Turing Machine in the role of language accepting device. *Next, we discuss how a TM can be used as a computer of functions*

Remark 1.7.1

For the purpose of discussing TMs as computers of functions, we make the following assumptions:

- A string ω over some alphabet say Σ will be written on the tape as $\#\omega\#$, where $\#$ is the blank symbol.
- Also initially, the TM will be scanning the *right-most* $\#$ of the string $\#\omega\#$.

Thus, the initial configuration, $(q_0, \#\omega\#)$ represents the starting point for the computation of the function with ω as input.

The assumption facilitates computation of composition of functions.

Though, most of the time, we require functions of one or more arguments having only integer values with values of arguments under the functions again as integers, yet, we consider functions with domain and codomain over *arbitrary* alphabet sets say Σ_0 and Σ_1 respectively, neither of which contains the blank symbol $\#$.

Next we define what is meant by computation, using Turing Machine, of a function

$$f: \Sigma_0^* \rightarrow \Sigma_1^*$$

Definition: A function $f: \Sigma_0^* \rightarrow \Sigma_1^*$ is said to be *Turing-Computable, or simply computable*, if there is a Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$, where Σ contains the following holds:

$$(q_0, \#\omega\#) \vdash_m^* (h, \#\mu\#)$$

whenever $\omega \in \Sigma_0^*$ and $\mu \in \Sigma_1^*$ satisfying $f(\omega) = \mu$.

Remark 1.7.2

It may be noted that, if the string ω contains some symbols from the set $\Sigma - \Sigma_0$, i.e, symbols not belonging to the domain of f , then the TM may hang or may not halt at all.

Remark 1.7.3

Next, we discuss the case of functions which require k arguments, where k may be any finite integer, greater than or equal to zero. For example, the operation PLUS takes two arguments m and n and returns $m + n$.

The function f with the rule
 $f(x, y, z) = (2x + y) * z$
 takes three arguments,

The function C with rule
 $C() = 17$
 takes zero number of arguments

Let us now discuss how to represent k distinct arguments of a function f on the tape. Suppose $k = 3$ and x_1, x_2, y_1, y_2, y_3 and z_1, z_2 are the three strings as three arguments of function f . If these three arguments are written on the tape as

#	x_1	x_2	y_1	y_2	y_3	z_1	z_2	#
---	-------	-------	-------	-------	-------	-------	-------	---

then the above tape contents may even be interpreted as a single argument viz. $x_1 x_2, y_1 y_2 y_3 z_1 z_2$. Therefore, in order, to avoid such an incorrect interpretation, the arguments are separated by #. Thus, the above three arguments will be written on the tape as

#	x_1	x_2	#	y_1	y_2	y_3	#	z_1	z_2	#
---	-------	-------	---	-------	-------	-------	---	-------	-------	---

In general, if a function f takes $k \geq 1$ arguments say $\omega_1, \omega_2, \dots, \omega_k$ where each of these arguments is a string over Σ_0 (i.e., each ω_i belongs to Σ_0^*) and if $f(\omega_1, \omega_2, \dots, \omega_k) = \mu$ for some $\mu \in \Sigma_1^*$; then we say f is Turing Computable if there is a Turing Machine M such that

$$(q_0, e, \# \omega_1 \# \omega_2 \dots \# \omega_k \#, e) \vdash_M^* (h, e, \# \mu \#, e)$$

Also, when f takes zero number of arguments and $f() = \mu$ then, we say f is computable, if there is a Turing Machine M such that

$$(q_0, e, \# \# \#, e) \vdash_M^* (h, e, \# \mu \#, e)$$

Remark 1.7.4

Instead of functions with countable, but otherwise arbitrary sets as domains and ranges, we consider only those functions, for each of which the domain and range is the set of natural numbers. This is not a serious restriction in the sense that any countable set can, through proper encoding, be considered as a set of natural numbers.

For natural numbers, there are various representations; some of the well-known representations are Roman Numerals (e.g. VI for six), Decimal Numerals (6 for six), Binary Numerals (110 for six). Decimal number system uses 10 symbols viz. 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Binary number system uses two symbols denoted by 0 and 1. In the discussion of Turing Computable Functions, the unary representation described below is found useful. The unary number system uses one symbol only:

Let the symbol be denoted by I then the number with name six is represented as $IIIIII$. In this notation, zero is represented by empty/null string. Any other number say twenty is represented in unary systems by writing the symbol I , twenty times. In order to facilitate the discussion, the number n , in unary notation will be denoted by I^n instead of writing the symbol I , n times.

The *advantage of the unary representation* is that, in view of the fact that most of the symbols on the tape are input symbols and if the input symbol is just one, then the *next state* will generally be determined by **only the current state**, because the other determinant of the next state viz tape symbol is most of the time the unary symbol.

We recall that for the set X , the notation X^* represents the set of all finite strings of symbols from the set X . Thus, any function f from the set of natural number to the set of natural numbers, *in the unary notation*, is a function of the form: $f: \{1\}^* \rightarrow \{1\}^*$

Definition: The function $f: \mathbb{N} \rightarrow \mathbb{N}$ with $f(n) = m$ for each $n \in \mathbb{N}$ and considered as $f: \{1\}^* \rightarrow \{1\}^*$, with $\{1\}$ a unary number system, will be called **Turing Computable function**, if a TM M can be designed such that M starting in *initial tape configuration*

1 1 1

with n consecutive 1's between the two #'s of the above string, halts in the following configuration

1 1 1

containing $f(n) = m$ 1's between the two #'s

The above idea may be further generalized to the functions of more than one integer arguments. For example, SUM of two natural numbers n and m takes two integer arguments and returns the integer $(n + m)$. The initial configuration with the tape containing the representation of the two arguments say n and m respectively, is of the form

1 1 ... 1 # 1 1 1

where the string contains respectively n and m 1's between respective pairs of #'s and Head scans the last #. The function SUM will be Turing computable if we can design a TM which when started with the initial tape configuration as given above, halts in the Tape configuration as given below:

1 1 ... 1 1 1

where the above string contains $n + m$ consecutive 1's between pair of #'s.

Example 1.7.5

Show that the SUM function is Turing Computable

The problem under the above-mentioned example may also be stated as: *Construct a TM that finds the sum of two natural numbers.*

The following design of the required TM, is not efficient yet explains a number of issues about which a student should be aware while designing a TM for computing a function.

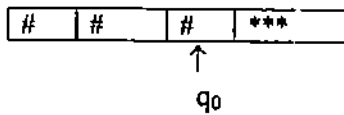
Legal and Illegal Configurations for SUM function:

In order to understand the design process of any TM for a (computable) function in general and that of SUM in particular, let us consider the possible *legal as well as illegal* initial configuration types as follows.

*Note: in the following, the sequence '...' denotes any sequence of 1's possibly empty and the sequences '***' denotes any sequence of Tape symbols possibly empty and possibly including #. Underscore denotes the cell being scanned.*

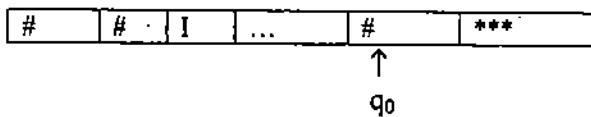
Legal initial configuration types:

Configuration (i)



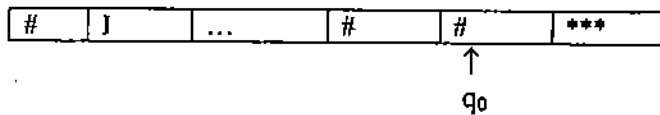
representing $n = 0, m = 0$

Configuration (ii)



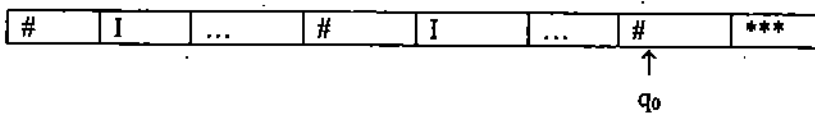
$n = 0, m \neq 0$

Configuration (iii)



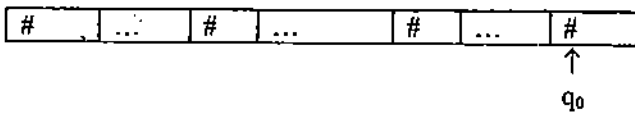
$n \neq 0, m = 0$

Configuration (iv)



$n \neq 0, m \neq 0$

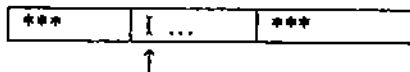
We treat the following configuration



containing two or more than two #'s to the left of # being scanned in initial configuration, as valid, where '...' denotes sequence of I's only.

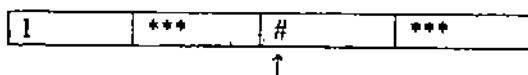
Some illegal initial configurations:

Configuration (v)



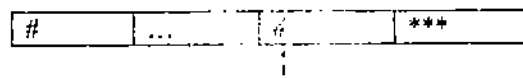
Where at least one of *** does not contain # and initially the Head is scanning an I or any symbol other than #. The configuration is invalid as it does not contain required number of #'s.

Configuration (vi), though is a special case of the above-mentioned configuration, yet it needs to be mentioned separately.



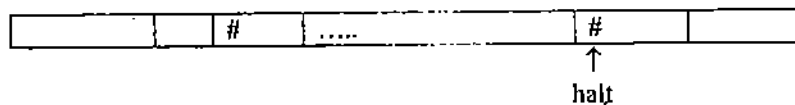
Left most symbol is I or any other non-# symbol
Where *** does not contain any #,

Configuration (vii)



Where *** does not contain # then the configuration represents only one of the natural numbers.

Also, in case of legal initial configurations, the final configuration that represents the result $m + n$ should be of the form.



with '.....' representing exactly $m + n$ 1's.

Also in case of illegal initial configurations, the TM to be designed, should be in one of the following three situations indicating non-computability of the function with an illegal initial input, as explained at the end of Section 1.5:

- (i) the TM has an infinite loop of moves;
- (ii) the TM Head attempts to fall off the left edge (i.e. the TM has Hanging configuration); or
- (iii) the TM does not have a move in a non-Halt state.

We use the above-mentioned description of initial configurations and the corresponding final configurations, in helping us to decide about the various components of the TM to be designed:

At this stage, we plan how to reach from an initial configuration to a final configuration. In the case of this problem of designing TM for SUM function, it is easily seen that for a legal initial configuration, we need to remove the middle # to get a final configuration.

- (a) **Summing up** initially the machine is supposed to be in the initial state (say) q_0
- (b) In this case of legal moves for TM for SUM function, first move of the Head should be to the *Left* only
- (c) In this case, initially there are at least two more #'s on the left of the # being scanned. Therefore, to keep count of the #'s, we must change state after scanning each #. Let q_1 , q_2 and q_3 be the states in which the required TM enters after scanning the three #'s
- (d) In this case the movement of the Head, after scanning the initial # and also after scanning one more # on the left, should continue to move to the *Left* only, so as to be able to ensure the presence of third # also. Also, in states q_1 and q_2 , the TM need not change state on scanning 1.

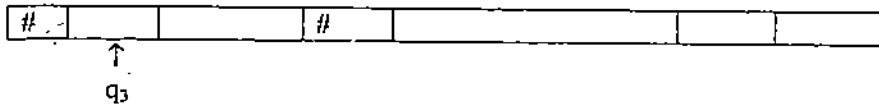
Thus we have

$$\begin{aligned} \delta(q_0, \#) &= (q_1, \#, L), \\ \delta(q_1, \#) &= (q_2, \#, L) \\ \text{and} \\ \delta(q_1, 1) &= (q_1, 1, L), \delta(q_2, 1) = (q_2, 1, L). \end{aligned}$$

However, from this point onward, the Head should *start moving to the Right*.

$$\therefore \delta(q_2, \#) = (q_3, \#, R).$$

Thus, at this stage we are in a configuration of the form



For further guidance in the matter of the design of the required TM, we again look back on the legal configurations.

- (c) In the configuration just shown above in q_3 , if the symbol being scanned is # (as in case of configuration (i) and configuration (ii)), then the only action required is to skip over I's, if any, and halt at the next # on the right.

However, if the symbol being scanned in q_3 of the above configuration, happens to be an I (as in case of configuration (iii) and configuration (iv)) then the actions to be taken, that are to be discussed after a while, have to be different.

But in both cases, movement of the Head has to be to the Right. Therefore, we need two new states say q_4 and q_5 such that

$$\delta(q_3, \#) = (q_4, \#, R)$$

(the processing/scanning argument on the left, is completed).

$$\delta(q_3, I) = (q_5, I, R)$$

(the scanning of the argument on the left, is initiated).

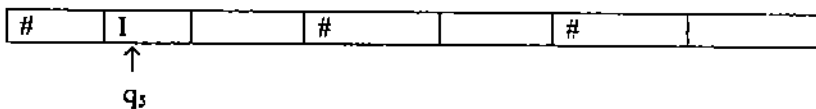
Taking into consideration the cases of the initial configuration (i) and configuration (ii) we can further say that

$$\delta(q_4, I) = (q_4, I, R)$$

$$\delta(q_4, \#) = (\text{halt}, \#, N)$$

Next, taking into consideration the cases of initial configuration (iii) and configuration (iv) cases, we decide about next moves including the states etc in the current state q_5 .

We are in the following general configuration (that subsumes the initial configuration (iii) and configuration (iv) cases)



Where the blank spaces between #'s may be empty or non-empty sequence of I's. Next landmark symbol is the next # on the right. Therefore, we may skip over the I's without changing the state i.e

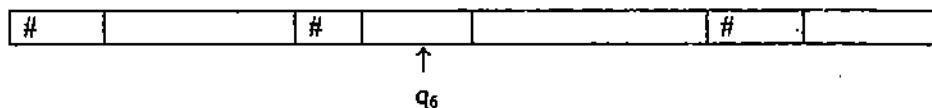
$$\delta(q_5, I) = (q_5, I, R)$$

But we must change the state when # is encountered in q_5 , otherwise, the next sequence of I's will again be skipped over and we will not be able to distinguish between configuration (iii) and configuration (iv) for further necessary action. Therefore

$$\delta(q_5, \#) = (q_6, \#, R)$$

(notice that, though at this stage, scanning of the argument on the left is completed, yet we can not enter in state q_4 , as was done earlier, because in this case, the sequence of subsequent actions have to be different. In this case, the # in the middle has to be deleted, which is not done in state q_4)

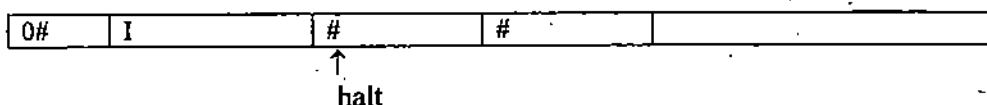
Thus, at this stage we have the general configuration as



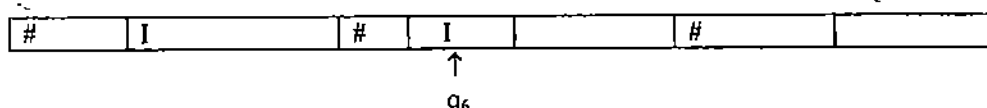
Next, in q_6 , if the current symbol is a #, as is the case in configuration (iii), then we must halt after moving to the left i.e.

$$\delta(q_6, \#) = (\text{halt}, \#, L)$$

we reach the final configuration



However, if we are in the configuration (iv) then we have



Then the following sequence of actions is required for deleting the middle #:

Action (i): To remove the # in the middle so that we get a continuous sequence of I's to represent the final result. For this purposes, we move to the left and replace the # by I. But then it will give one I more than number of I's required in the final result.

Therefore

Action (ii): We must find out the rightmost I and replace the rightmost I by # and stop, i.e. enter halt state. In order to accomplish Action (ii) we reach the next # on the right, skipping over all I's and then on reaching the desired #, and then move left to an I over there. Next, we replace that I by # and halt.

Translating the above actions in terms of formal moves, we get

For Action (i)

$$\begin{aligned} \delta(q_6, I) &= (q_7, I, L) \\ \delta(q_7, \#) &= (q_8, I, R) \end{aligned}$$

(at this stage we have replaced the # in the middle of two sequences of I's by an I)

For Action (ii)

$$\begin{aligned} \delta(q_8, I) &= (q_8, I, R) \\ \delta(q_8, \#) &= (q_9, \#, L) \\ \delta(q_9, I) &= (\text{halt}, \#, N) \end{aligned}$$

It can be verified that through above-mentioned moves, the designed TM does not have a next-move at some stage in the case of each of the illegal configurations.

Formally, the SUM TM can be defined as:

$$\text{SUM} = (Q, \Sigma, \Gamma, \delta, q_0, h)$$

where $Q = \{q_0, q_1, \dots, q_{10}, \text{halt}\}$

$$\begin{aligned} \Sigma &= \{I\} \\ \Gamma &= \{I, \#\} \end{aligned}$$

and

the next-move (partial) function δ is given by the Table

Turing Machine

	I	#
q_0	-	$(q_1, \#, L)$
q_1	(q_1, I, L)	$(q_2, \#, L)$
q_2	(q_2, I, L)	$(q_3, \#, R)$
q_3	(q_3, I, R)	$(q_4, \#, R)$
q_4	(q_4, I, R)	$(\text{halt}, \#, N)$
q_5	(q_5, I, R)	$(q_6, \#, R)$
q_6	(q_7, I, L)	$(\text{halt}, \#, L)$
q_7	-	(q_8, I, R)
q_8	(q_8, I, R)	$(q_9, \#, L)$
q_9	$(\text{halt}, \#, N)$	
halt	-	-

'-' indicates that δ is not defined

Remark 1.7.6

As mentioned earlier also in the case of design of TM for recognizing the language of strings of the form $b^n d^n$, the design given above contains too detailed explanation of the various steps. The purpose is to explain the involved design process in fine details for better understanding of the students. However, the students need not supply such details while solving a problem of designing TM for computing a function. While giving the values of Q, Σ, Γ explicitly and representing δ either by a table or a transition diagram, we need to give only some supporting statements to help understanding of the ideas involved in the definitions of Q, Σ, Γ and δ .

Example 1.7.7

Construct a TM that multiplies two integers, each integer greater than or equal to zero (Problem may also be posed as: show that multiplication of two natural numbers is Turing Computable)

Informal Description of the solution:

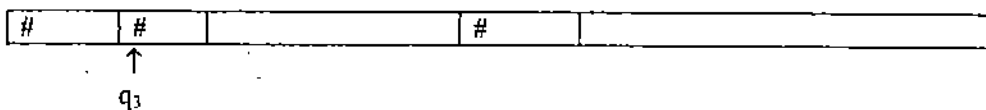
The legal and illegal configurations for this problem are the same as those of the problem of designing TM for SUM function. Also, the moves required to check the validity of input given for SUM function are the same and are repeated below:

- $\delta(q_0, \#) = (q_1, \#, L)$
- $\delta(q_1, \#) = (q_2, \#, L)$
- $\delta(q_1, I) = (q_1, I, L)$
- $\delta(q_2, \#) = (q_3, \#, R)$
- $\delta(q_2, I) = (q_2, I, L)$

Next, we determine the rest of the behaviour of the proposed TM.

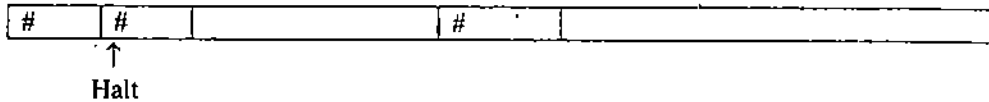
Case I

When $n = 0$ covering configuration (i) and configuration (ii) The general configuration is of the form



To get representation of zero, as, one of the multiplier and multiplicand is zero, the result must be zero. We should enter state say q_4 which skips all 1's and meets the next # on the right.

Once the Head meets the required #, Head should move to the left replacing all 1's by #'s and halt on the # it encounters so that we have the configuration

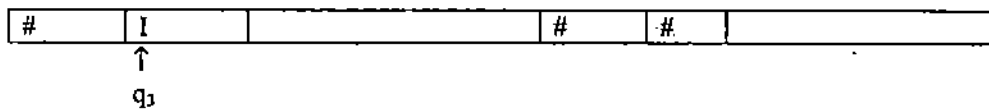


The moves suggested by the above explanation covering configuration (i) and configuration (ii) are:

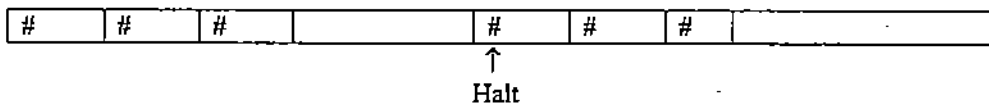
- $\delta(q_3, \#) = (q_4, \#, R)$
- $\delta(q_4, I) = (q_4, I, R)$
- $\delta(q_4, \#) = (q_5, \#, L)$
- $\delta(q_5, I) = (q_5, \#, L)$
- $\delta(q_5, \#) = (Halt, \#, R)$

Case II

Covering configuration (iii), we have at one stage

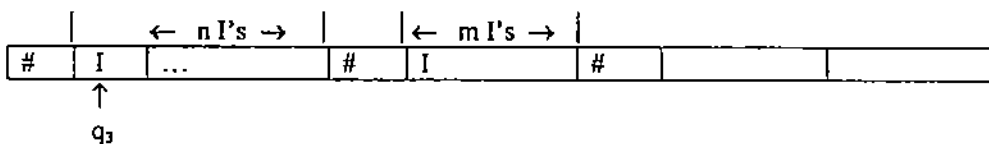


If we take $\delta(q_3, I) = (q_4, \#, R)$, then we get the following desired configuration in finite number of moves:

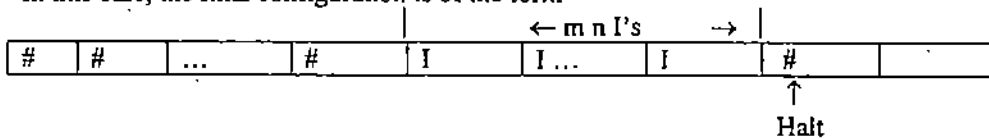


Case III

While covering the configuration (iv), At one stage, we are in the configuration



In this case, the final configuration is of the form

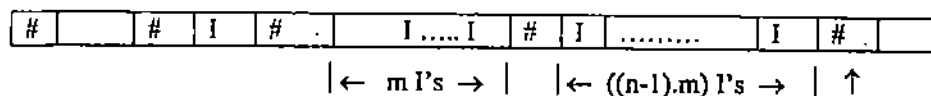


The strategy to get the representation for n m I's consists of the following steps

- (i) replace the left-most I in the representation of n by # and then copy the m I's in the cells which are on the right of the # which was being scanned in the initial configuration. In the subsequent moves, copying of I's is initiated in the cells which are in the left-most cells on the right hand of last I's on the tape, containing continuous infinite sequence of #'s.

Repeat the process till all I's of the initial representation of n, are replaced by #. At this stage, as shown in the following figure, the tape contains m I's of the initial representation of the integer m and additionally n.m I's. Thus the tape contains m extra #'s than are required in the representation of final result. Hence, we replace all I's of m by #'s and finally skipping over all I's of the representation of (n . m) we reach the # which is on the right of all the (n . m) I's on the tape as required.

Alternatively: In stead of copying n times of the m I's, we copy only (n-1) times to get the configuration



Then we replace the # between two sequences of I's by I and replace the right-most I by # and halt.

The case of illegal initial configurations may be handled on similar lines as were handed for SUM Turing machine

Remark 1.7.8

The informal details given above for the design of TM for multiplication function are acceptable as complete answer/solution for any problem about design of a Turing Machine. However, if more detailed formal design is required, the examiner should explicitly mention about the required details.

Details of case (iii) are not being provided for the following reasons

- (i) Details are left as an exercise for the students
- (ii) After some time we will learn how to construct more complex machines out of already constructed machines, starting with the construction of very simple machines. One of the simple machines discussed later is a *copying machine* which copies symbols on a part of the tape, in other locations on the tape.

Ex. 7) Design a TM to compute the binary function MONUS (or also called PROPER SUBTRACTION) defined as follows:

$$\text{Monus} : N \times N \rightarrow N$$

(Note 0 also belongs to N)

such that

$$\text{monus}(m, n) = \begin{cases} m-n & \text{if } m \geq n \\ 0 & \text{else} \end{cases}$$

Ex.8) To compute the function $n \pmod 2$

Let f denotes the function, then

$$f: N \rightarrow \{0, 1\}$$

is such that

$$f(n) = \begin{cases} 0 & \text{if } n \text{ is even,} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

1.8 MODULAR CONSTRUCTION OF COMPLEX TURING MACHINES

In the previous example of constructing a Turing Machine even for a simple task of multiplying two numbers, we saw construction was quite complex. The handling of complexity can be attempted by looking at the total machine in terms of sub-machines.

In this section, we look at the task of constructing complex Turing Machines by suitably combining already constructed simpler Turing Machines. For this purpose, we discuss some *Basic Machines and Rules* for combining already constructed machines into more complex machines. Also, we develop notation for expressing the involved rules and denoting the process for combining.

We begin by giving below rules of combining Turing Machines to get more complex TMs from the already constructed Turing Machines. Let M be the TM which is to be constructed by combining the already constructed machines viz. M_1, M_2, \dots, M_k , where $M_i = \{Q_i, \Sigma_i, \Gamma_i, \delta_i, q_{0i}, h_i\}$ and $M = \{Q, \Sigma, \Gamma, \delta, q_0, h\}$ and M will start its actions in the machine M_1 .

Then the rules for constructing M out of M_i are:

Rule 1: Assume all the sets $Q_1, Q_2 \dots Q_k$, are all mutually disjoint sets. If there is an overlap, then rename the elements of some sets so that all the sets are mutually disjoint.

Rule 2: The state q_{0i} , the initial state of M_i will be the initial state of M i.e. $q_0 = q_{01}$; however, the initial state status of q_{02}, \dots, q_{0k} is removed. Also, the set of states for M will contain as its subset each of Q_i for $i = 1, 2, \dots, k$.

Rule 3: The halt-state status of each $h_i, i = 1, 2, \dots, k$ is removed and a new state h is included in Q which will serve as the halt state of M . However, each h_i remains a state of M , but its status as halt state is removed.

Thus

(In the following \cup denotes set union)

$$Q = \bigcup_{i=1}^k Q_i \cup \{h\},$$

$$\text{where } h \notin \bigcup_{i=1}^k Q_i,$$

Rule 4: Σ contains $\bigcup_{i=1}^k \Sigma_i$ and

$$\Gamma \text{ contains } \bigcup_{i=1}^k \Gamma_i.$$

It may be noted that Σ may contain some more symbols, in addition to the symbols in $\bigcup_{i=1}^k \Sigma_i$, and similarly Γ may contain some more symbols, in addition to the symbols in

$$\bigcup_{i=1}^k \Gamma_i,$$

Rule 5(i): If the composite machine M is to halt on reaching h_i with symbol currently being scanned as x , then introduce a move $\delta(h_i, x) = (h, x, N)$.

Rule 5(ii) If, in stead of halting in the state h_i , of machine M_i , while scanning the symbol x , the composite machine M is required to transfer the control to some machine say $M_b = \{Q_b, \Sigma_b, \Gamma_b, \delta_b, q_{0b}, h_b\}$ in some state say p and the symbol x is required to be replaced by z then introduce the move $\delta(h_i, x) = (p, z, N)$. Diagrammatically we have

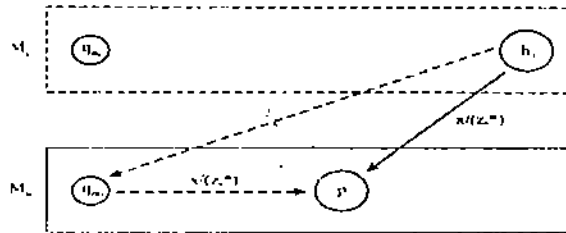


Fig. 1.8.1

This completes the details of the general rules for obtaining a composite machine out of already designed machines as components. However, there may be some special rules for design of each particular composite machine.

Example 1.8.1

Let $M_i = \{Q_i, \Sigma_i, \Gamma_i, \delta_i, q_i, h_i\}$ for $i = 1, 2$
 be two given TMs. We are required to construct a TM which first simulates M_1 and then M_2 and halts.

Then M is obtained by taking

$$M = (Q, \Sigma, \Gamma, \delta, q, h)$$

Where

$$Q = Q_1 \cup Q_2 \cup \{h\}$$

$$\Sigma = \Sigma_1 \cup \Sigma_2, \quad \Gamma = \Gamma_1 \cup \Gamma_2, \quad q = q_1 \text{ and } h = h_2$$

and δ consists of

- (i) all the moves defined by δ_1
- (ii) all the moves defined by δ_2
- (iii) $\delta(h_1, x) = (q_2, x, N)$ for all $x \in \Gamma$ (where q_2 is the initial state of M_2)

In words: M is obtained by

- (i) taking initial state q_1 of M_1 as initial state of M
- (ii) removing halt state status of h_1 of M_1 and initial state status of q_2 of M_2
- (iii) Introducing δ moves from the (old) halt state h_1 of M_1 to be (old) initial state q_2 of M for each symbol x of the tape s.t.

$$\delta(h_1, x) = (q_2, x, N)$$

Diagrammatically M is given by

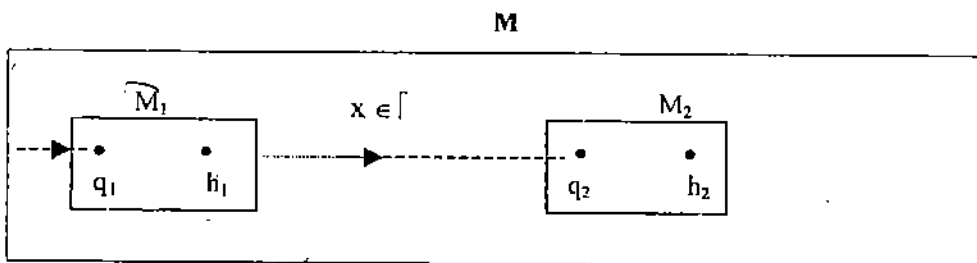


Fig. 1.8.2

Example 1.8.2

Let us consider one way of combining the following three machines. (There are many possible ways of combining these three machines). For all the three machines, the input symbol set $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, \#\}$ are the same. Further.

$M_1 = (Q_1, \Sigma, \Gamma, \delta_1, q_{10}, h_1)$, which finds the first 1 after the current symbol and halts, and is given by $Q_1 = \{q_{10}, q_{11}, h_1\}$ with

- $\delta_1(q_{10}, x) = (q_{11}, x, R)$ for each of $x = 0, 1$ and $\#$.
- $\delta_1(q_{11}, 0) = (q_{11}, 0, R)$,
- $\delta_1(q_{11}, \#) = (q_{11}, \#, R)$; and
- $\delta_1(q_{11}, 1) = (h_1, 1, N)$

$M_0 = (Q_0, \Sigma, \Gamma, \delta_0, q_{00}, h_0)$, which finds the first 0 after the current symbol and halts, is given by $Q_0 = \{q_{00}, q_{01}, h_0\}$ with

- $\delta_0(q_{00}, x) = (q_{01}, x, R)$ for each of $x = 0, 1$ and $\#$
- $\delta_0(q_{01}, 1) = (q_{01}, 1, R)$,
- $\delta_0(q_{01}, \#) = (q_{01}, \#, R)$; and
- $\delta_0(q_{01}, 0) = (h_0, 0, N)$

$M_3 = (Q_3, \Sigma, \Gamma, \delta_3, q_3, h_3)$, which moves the tape Head one cell to the right and Halts where

- $Q_3 = \{q_{30}, h_3\}$
- $\delta_3(q_{30}, x) = (h_3, x, R)$ for each of $x = 0, 1$ or $\#$.

Now we combine the above three Turing Machines M_1 , M_2 and M_3 as building blocks, so that the constructed composite Machine M finds the first occurrence of a non-blank symbol (i.e., symbol which is a 0 or a 1) after skipping two symbols, viz, currently being scanned symbol and the immediately next symbol. For example, the composite machine returns

- (i) 1 for each of the following input strings
0###10# or
00##10# or
001#00
- (ii) 0 for each of the strings
11#0 or
11###0 or
110##1

The Turing Machine M is given by

$M = (Q, \Sigma, \Gamma, \delta, q_{03}, h)$, where
 $Q = \{q_{00}, q_{01}, h_0, q_{10}, q_{11}, h_1, q_{30}, h_3, h\}$

In the machine M , q_{00} and q_{10} are not initial states. Also h_1, h_2, h_3 , are no more halt states and h is the new Halt State.

In addition to simulating moves of M_0, M_1 and M_3 , the following moves are added:

- $\delta(h_3, *) = (q_{00}, *, N)$
- $\delta(h_3,) = (q_{10}, , N)$, where $**$ is any symbol from Γ .

so that from M_3 , we may go to M_0 or M_1 on scanning any symbol.
 Further in order to halt in the new machine, we introduce

$$\delta(h_0, *) = (h, *, N)$$

$$\delta(h_1, *) = (h, *, N), \text{ where } "*" \text{ is any symbol from } \Gamma,$$

Note: The constructed machine is of Non-Deterministic (to be defined) type.
 After appropriate shortcut Notations, the combined TM is graphically as shown below:

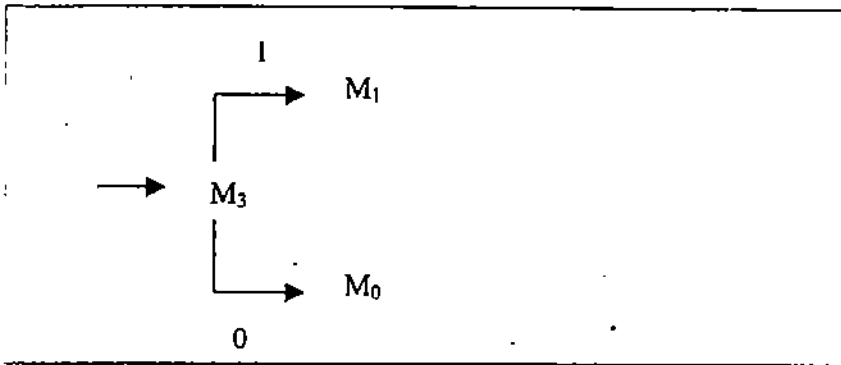


Fig. 1.8.3

Some Short cut Notations:

- (i) If there is the same output and same next state for more than one inputs in a particular state, then single labeled arrow may be used instead of more than one arrow, e.g., The part of the transition diagram

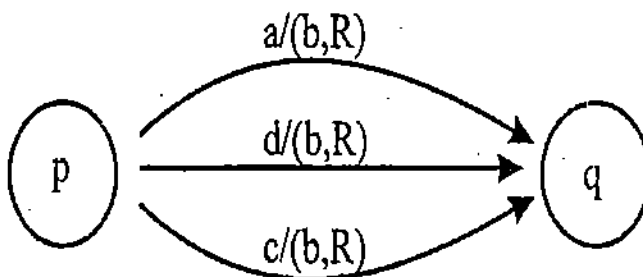


Fig. 1.8.4

may be replaced by

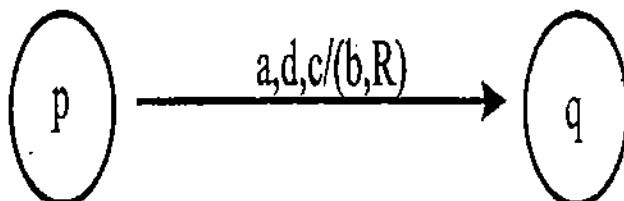


Fig. 1.8.5

- (ii) Further, in the case discussed above, if $\Gamma = \{a, b, c, d\}$, is the set of tape symbols, then the diagram may be further modified as

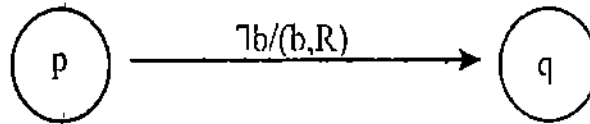


Fig. 1.8.6

where $\bar{\Gamma}$ denotes 'except for b, on all other tape symbols'.

The same shorthand is used when instead of states p and q in the two figures above, we have component machines M_1 and M_2 .

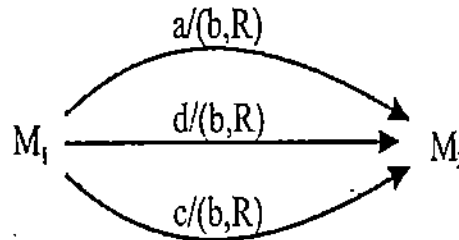


Fig. 1.8.7

Further, if on all inputs the composite machine operates as machine M_1 until M_1 halts, and then M_2 and then operates as M_2 would operate, then the following notation may be used.

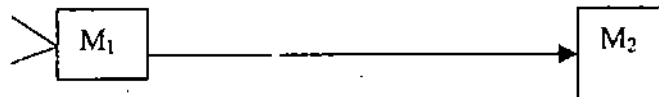


Fig. 1.8.8

Where there are no labels on the arrow.

- (iii) If the composite machine M is such that first it operates as machine M_1 until it halts and then operates as say M_2 or M_3 depending on the symbol being scanned at the time of halting of M_1 , say out of a or b respectively, then the following notation is used.

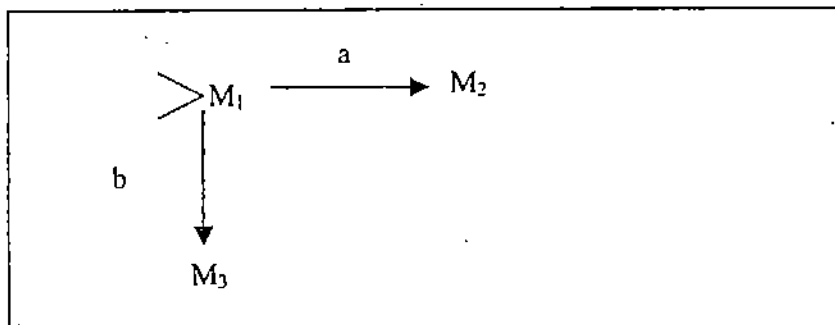


Fig. 1.8.9

In the case of above composition of machines, in addition to all moves defined by δ_1 and δ_2 for machines M_1 and M_2 we have the additional moves:

(i) $\delta(h_1, a) = (q_2, a, N)$ and

(ii) $\delta(h_1, b) = (q_3, b, N)$,

where h_1 is the halt state of M_1 and q_2 and q_3 are the initial states of M_2 and M_3 respectively.

Some Basic Machines and Notations:

As the purpose is to explain how complex machines are obtained combining basic machines, the basic machines do not necessarily start scanning the left-most symbol.

There are two types of basic machines viz.

(i) **Symbol Writing Machines:** Let $M = (Q, \Sigma, \sqsupset, \delta, q_0, h)$ where and let $a \in \Sigma$ be a particular symbol such that for some $\delta(q_0, x) = (h, a, N)$ for all $x \in \Sigma$ (where x is used in the sense of a variable, which actually is not a member of Σ).

This machine after starting in the initial state q_0 and reading any symbol, writes 'a' in place of the current symbol and halts.

We denote such machines by W_a or sometimes just by a

Where a may denote the symbol a as well as the machine that writes a.

However, context will resolve whether a particular occurrence denotes the symbol or the machine.

(ii) **Right/Left head Moving Machines:**

(a) **Right Head Moving Machine**

Let $M = (Q, \Sigma, \sqsupset, \delta, q_0, h)$ and δ is given by

$\delta(q_0, x) = (h, x, R)$ for all $x \in \Sigma$

(where x is used in the sense of a variable, which actually is not a member of Σ).

This machine in the initial state q_0 scans the current symbol and whatever may be the current symbol, moves Head one square to the Right and halts.

Such a machine is denoted by R .

(b) **Left Head Moving Machine**

Similarly, if a machine in the initial state q_0 , scans the current symbol, and whatever may be the current symbol, it moves Head one square to the left and then halt;

such a machine is denoted by L .

(c) A machine which goes on moving to the Right except when it meets a specific symbol say $a \in \Sigma$, and on meeting a , the machine halts. Such a machine is denoted by

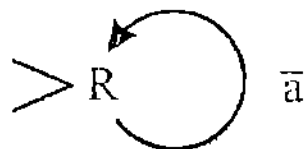


Fig. 1.8.10

Where \bar{a} denotes any symbol from $\Sigma - \{a\}$

Or is denoted by

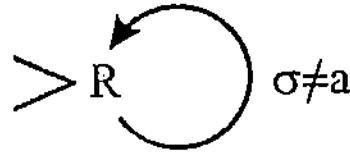


Fig. 1.8.11

Or is denoted by just

R_a

(note in R_a there is no bar on a).

Therefore R_a finds the occurrence of first a to the right and halts.

- (d) Thus $R_{\bar{a}}$ denotes the machine, which finds the first non-blank symbol on the right.

In general, $R_{\bar{a}}$ denotes the machine which while moving to the right skips all a 's and halts on finding a symbol different from a

- (e) L_a finds the occurrence of the first a to the left and halts. $L_{\bar{a}}$ denotes the machine, which finds the first non-blank symbol on the left and halts.

In general $L_{\bar{a}}$ denotes the machine which while moving to the left skips all a 's and halts on finding a symbol different from a . On scanning the symbol a , the machine halts. Such a machine may be denoted by either of the following three notations:

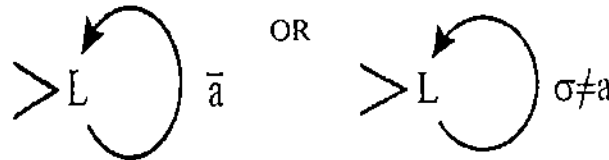


Fig. 1.8.12

1.8.13

Using the above notation and basic machines we provide notation for more complex machines.

Example 1.8.3

$> R \xrightarrow{a} R$ denotes a machine, which in the initial state moves the Head one square to the Right and halts if the new symbol being scanned is *not* a . However, if the new symbol being scanned is a then, the Head moves Right once more.

Remark 1.8.4

- (i) We should note the difference between R_a and Ra (and similarly L_a and La) R_a denotes the machine that finds the first a on the Right. But Ra denotes a machine which first moves to the right and then writes ' a ' in place of the new symbol being scanned. Further,

- (ii) the sequence like $R a R b L$ denotes a combination of five machines, the first of which moves the Head to the Right and halts; then second machine writes an a in the current cell and halts; then the third machine again moves the Head to the Right and halts; then the fourth machine writes 'b' in the current cell and halts; and then finally the last machine moves the Head to the Left and halts. Thus, if initially the Tape configuration is as follows:

..... c b a b d # a c b #

Then after all the actions of the above-mentioned combined machine, the Tape configuration will be

c b a a b # a c b #

However, the combined machine $R_a R_b L$ when starts in the same Tape configuration, viz.,

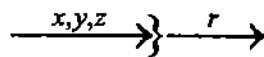
c b a b d # a c b #

will yield

c b a b d # a a b #.

$R_a R_b L$ first searches for the next a to the right on the Tape through the machine R_a . Then R_a machine halts but R_b machine initiates and moves to the first b on the right and halts. Then the machine L initiates and moves the Head one cell to the Left.

Another Short-Hand: We use the notation



to denote that when the current symbol is any one of x, y or z then the machine should proceed in the direction of the arrow with r representing the symbol which is actually present

For example, M is the composite machine

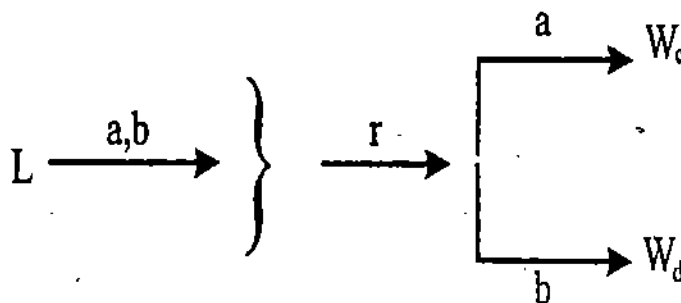


Fig. 1.8.14

and Tape configuration is

c b a d e ...

Then the machine M , first moves the Head to the Left, and it finds 'b' there and hence activates the machine w_d , which writes d in place of b and halts. Thus the configuration after M has executed and halted will be

c d a d e ...

Using the shorthand notation introduced above, we describe a number of Turing Machines. Some of these machines would be quite useful in the construction of more complex machines and hence will be given standard names.

Example 1.8.5

S_R The right-shifting machine. The machine takes an input of the form

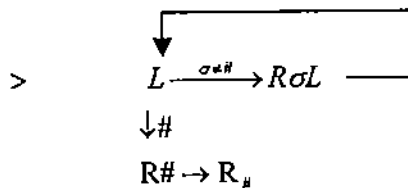
a b c b a # #
 and returns
 # # a b c b a # #
 (with one extra, # on the left hand side)

First, we explain the strategy behind the construction of the machine S_R.

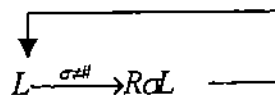
From the current position, we move to the cell on left and note the symbol over there. And if it is not # then copying it in the cell to the right of the cell of the noted element; i.e., we apply $L \xrightarrow{\sigma \neq \#} R \sigma L$. The process is repeated unless the noted symbol is #. The process terminates on encountering #, followed by moving to the Right and writing # over there and then moving from there to the # on the right of given sequence of non-blank symbols. We may further explain the meaning of the expression

$$L \xrightarrow{\sigma \neq \#} R \sigma L$$

In the above expression L means, first move to the Left. Then $\xrightarrow{\sigma \neq \#}$ means note the symbol over there and call that symbol σ . If the noted symbol which we call σ is not # then execute R σ L. Otherwise take some other action denoted by a different arrow, if any Else stop. Next R σ L denotes that first move to the right, write down the symbol which was noted down earlier which we call σ and then move left again. Therefore S_R is of the form



Let us call execution of the following loop, starting with left-most L as one iteration.



Then we explain the effect of each iteration as follows:

Let us start in the configuration.

a b c b a #

Then after one iteration we reach (just before the beginning of the left-most L)

a b c b a a

and after next iteration we reach the configuration

a b c b b a

Then we have

a b c c b a

Next, we have

a a b c b a

At this stage when S_R applies L the tape is of the configuration.

a a b c b a

Therefore, the branch $R\#$ is taken up.

i.e. we get

$\# \# a b c b a \#$

And, finally, when $R\#$ is executed, then we get the configuration $\# \# a b c b a \#$.

Example 1.8.6

To construct the *copy machine C* which takes a string of the form $\#\omega\#$ in the initial state and gives, in the halt state, the configuration $\#\omega\#\omega\#$ where ω is a string of tape symbols but not containing the blank symbol $\#$.

First we explain how the proposed machine should work *through an example and side by side*, be as given below give the construction of C. Let initially, we be in the configuration

$\# b a c b c c a \# \# \dots$

Step I Move to the $\#$ which is on the left of the sequence of non-blank symbols. In other words we apply $L\#$.

After this step we would be in the configuration

$\# b a c b c c a \# \# \dots$

(i.e. first component machine would be $L\#$)

Step II Next we move right and note the symbol (*in this case b*) and replace it by $\#$ and **cross over** all non-blank symbols and first $\#$ on the Right to reach the second $\#$ on the right of non-blank symbols i.e. we have the configurations $\# \# a c b c c a \# \#$ and we remember b also through σ .

We write this b in place of $\#$ being scanned, to get the configuration

$\# \# a c b c c a \# \underline{b}$. This step may be summarized as

$$R \xrightarrow{\sigma \#} \# R\# R\# \sigma$$

Step III. Then we should come back to the original position of b through $L\#$ and write back b. Thus, we reach the configuration

$\# \underline{b} a c b c c a \# b \# \dots$

The machine component of Step III is given by

$$L\# L\# \sigma$$

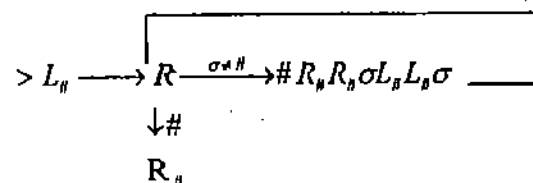
Iterative Steps

Now copying of next symbol (*which is 'a' in this case*) can be carried out by applying the Step II followed by Step III once again.

Final Steps

The copying process should stop when we encounter $\#$, after a finite number of repetitions of 'Step II and Step III. At this stage we should move to the $\#$ which is first on the right of the given string.

Thus the copying machine C is as given below:



To have better understanding, we consider the traces of some more iterations.

After second iteration of Step II and Step III, the tape configuration is

$$\# b a c b c c a \# b a \# \# \dots$$

After 7 iterations we get

$$\# b a c b c c a \# b a c b c c a \#$$

As the copying machine finally scans the # following the copied part through the last component R_k of the copying machine, is justified, in view of keeping the Head on the #, which is to the right of all non-blanks.

Finally, we get the configuration

$$\# b a c b c c a \# b a c b c c a \#$$

Example 1.8.7

Design a Turing Machine that decrements one from a positive integer, using binary representation for integers.

Solution: In order to construct the desired machine, we consider some cases of Tape configurations representing the binary numbers before and after subtraction of 1.

Case (i) When the given binary number is represented on the tape in the form

$$\# x_1 \dots x_k 1 \#$$

where $x_1 = 1$ and x_i may be 0 or 1 $i = 2, 3, \dots, k$, then after subtraction of 1, the representation of the number becomes

$$\# x_1 \dots x_k 0 \#$$

requiring the change to only the right-most bit.

Case (ii) When the given binary number is represented on the tape in the form

$$\# x_1, \dots, x_k 1 0 \#$$

then after subtraction the binary number representation becomes

$$\# x_1 \dots x_k 0 1 \#$$

requiring the two least significant bits to be reversed.

Case (iii) When the given binary number is represented on the tape in the form

$$\# x_1 \dots x_k 1 \underbrace{0 0 \dots 0}_i \#$$

The number after subtraction of 1 is given by

$$\# x_1 \dots x_k 0 \underbrace{1 1 \dots 1}_i \#$$

$$\text{Thus } \underbrace{1 0 0 \dots 0}_i \text{ zeros is replaced by } 0 \underbrace{1 1 1 1}_i \text{ ones}$$

Thus in case (iii), which is a generalization of case (ii), each of all the continuous zeros from right to left, is replaced by a 1 and the 1, on the left of these 0's is replaced by a 0.

Case (iv) is again a special case of case (iii), in which the given binary number is represented in the form

1 0 0 ... 0

then after subtraction of 1 we get the binary number representation of the form

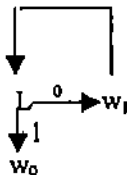
0 1 1 1 ... 1

However, in our binary representations, leading bit, i.e., left-most bit is always 1. Therefore, we need to delete the leading 0, by shifting the string '0 1 1 1 ... 1 #' to the left so that we get # 1 1 1 1 ... 1 #

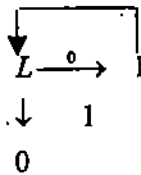
The process of subtraction of 1 from a binary number may now be summarized as follows:

Step I: The machine starts in the following configuration # # x_1 x_2 ... x_n # where $x_1 = 1$ and $x_i = 0$ or 1 for $i = 2, 3, \dots$

Step II: In view of the above case analysis, we attempt to find first 1 while moving from right to left and changing each of the 0 on the way to a 1. And when the Head scans the first 1, we change this 1 to a 0. This part of the machine may be represented by



Where w_i denotes 'write i' which can also be denoted by just i, i.e., the above diagram may be denoted as:



Step III: Next step is to remove the leading zero, if any, by shifting the rest of the binary string to the Left. This situation may occur if the initial tape configuration is

1 0 0 ... 0

resulting in the configuration

0 1 1 ... 1

In order that the representations is appropriate, having finally the most significant bit as 1, we shift the rest of the string to the left so that finally the tape configuration is of the form

1 1 ... 1

In order to execute Step III, we use L_1 so that configuration becomes

0 1 1 ... 1

Then move right and check the bit. If the bit is a 1 then we move to right to the next # through R_1 . If the bit is a 0, then we execute the following steps:

Case III (i) Write a # over 0 to get

$$\# \# 1 1 1 \dots 1 \#$$

Then we use S_L so that we get $\# 1 1 \dots 1 \#$.

Step III may be summarized as the machine

$$\begin{array}{c} L_0 R \xrightarrow{1} R_0 \\ \downarrow 0 \\ \downarrow \\ \# S_L \end{array}$$

Combing the machines of Step I, Step II and Step III

$$\begin{array}{c} \begin{array}{c} \boxed{} \\ \downarrow L \xrightarrow{0} 1 \end{array} \\ \downarrow 1 \\ 0 \\ \downarrow \\ L_0 R \xrightarrow{1} R_0 \\ \downarrow 0 \\ \# S_L \end{array}$$

Ex.9) Construct the machine S_L which transforms a string $\# \omega \#$ to $\omega \#$, i.e., shifts each element of ω one position to the Left.

Ex.10) To construct a Turing Machine which simulates a function
 $f: \Sigma^* \rightarrow \Sigma^*$ s.t.
 if $\omega \in \Sigma^*$ i.e. (i.e., ω is of the form $\omega = a_1 a_2 \dots a_k$ where $a_i \in \Sigma$)
 Then $f(\omega) = \omega \omega$
 i.e. if $\omega = a_1, a_2, \dots, a_k$ then f maps the configuration
 $\# a_1 a_2 \dots a_k \#$ to the configuration
 $\# a_1 a_2 \dots a_k a_1 a_2 \dots a_k \#$

Ex. 11) Design a TM that checks for palindromes over an alphabet $\{c, d\}$. In other words, if $\Sigma = \{c, d\}$ and $w \in \Sigma^*$, then the TM returns y for 'Yes' if $w = w^R$ and returns N for 'No' if $w \neq w^R$.

1.9 SUMMARY

In this unit, after giving informal idea of what a Turing machine is, the concept is formally defined and illustrated through a number of examples. Further, it is explained how TM can be used to compute mathematical functions. Finally, a technique is explained for designing more and more complex TMs out of already designed TMs, starting with some very simple TMs.

1.10 SOLUTIONS/ANSWERS

Exercise 1: The transition diagram of the required TM is as shown below:

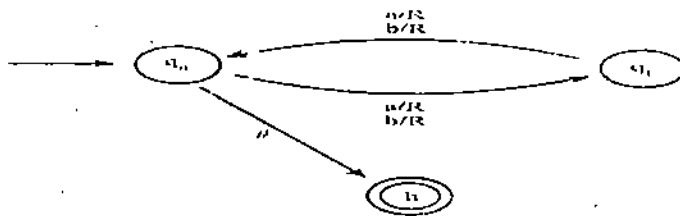


Fig.1.10.1

The required TM $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ with

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

$\Sigma = \{a, b\}$ and $\Gamma = \{a, b, \#\}$.

The next move function δ is given by the transition diagram above. If the input string is of even length the TM reaches the halt state h . However, if the input string is of odd length, then TM does not find any next move in state q_1 indicating rejection of the string.

Exercise 2: The transition diagram of the required TM is as shown below,

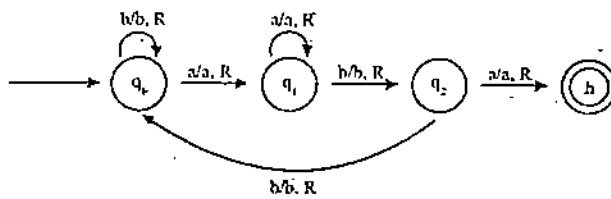


Fig. 1.10.2

The required TM $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ with

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

$\Sigma = \{a, b\}$ and $\Gamma = \{a, b, \#\}$.

The next move function is given by the transition diagram above.

The transition diagram almost explains the complete functioning of the required TM.

However, it may be pointed out that, if a string is not of the required type, then the blank symbol $\#$ is encountered either in state q_0 or in state q_1 or in state q_2 . As there is no next move for $(q_0, \#)$, $(q_1, \#)$ or $(q_2, \#)$, therefore, the string is rejected.

Exercise 3: The transition diagram of the required TM is as shown below:

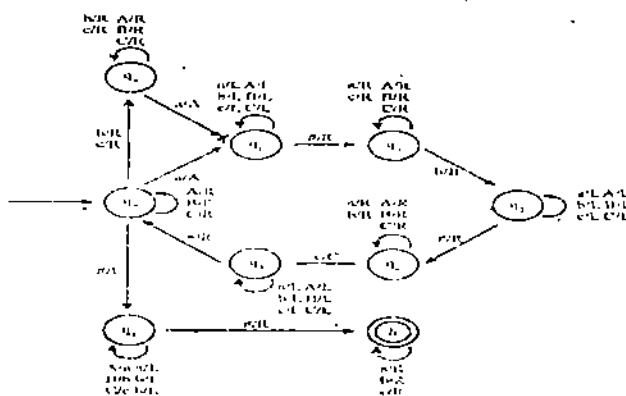


Fig. 1.10.3

The required TM $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ with

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, h\}$
 $\Sigma = \{a, b, c\}$ and $\Gamma = \{a, b, c, A, B, C, \#\}$ δ is shown by the diagram.

The design strategy is as follows:

Step I While moving from left to right, we find the first occurrence of a if it exists. If such an a exists, then we replace it by A and enter state q_1 , either directly or after skipping b's and c's through state q_4 .

In state q_1 , we move towards left skipping over all symbols to reach the leftmost symbol of the tape and enter state q_5 .

In q_5 , we start searching for b by moving to the right skipping over all non-blank symbols except b and if such b exists, reach state q_2 .

In state q_2 , we move towards left skipping over all symbols to reach the leftmost symbol of the tape and enter q_6 .

In q_6 , we start searching for c by moving to the right skipping over all non-blank symbols except c and if such c exists, reach state q_3 .

In state q_3 , we move towards left skipping all symbols to reach the leftmost symbol of the tape and enter state q_0 .

If in any one of the states q_4, q_5 or q_6 no next move is possible, then reject the string. Else repeat the above process till all a's are converted to A's, all b's to B's and all c's to C's.

Step II is concerned with the restoring of a's from A's, b's from B's and c's from C's, while moving from right to left in state q_7 and then after successfully completing the work move to halt state h.

Exercise 4: The Transition Diagram of the TM that recognizes strings of the form $b^n d^n$, $n \geq 1$ and designed in the previous section is given by the following diagram.

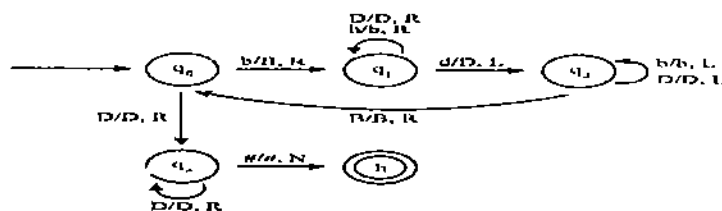


Fig. 1.10.4

Exercise 5: The transition diagram of the required TM is as shown below.

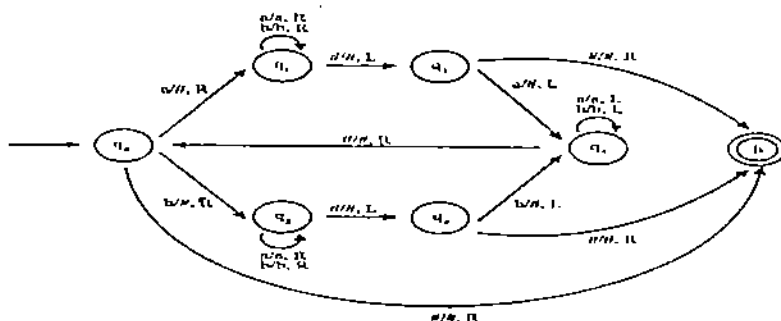


Fig. 1.10.5

The required TM $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ with

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

$\Sigma = \{a, b\}$ and $\Gamma = \{a, b, \#\}$.

The next move function is given by the transition diagram above.

The proposed TM functions as follows:

- (i) In state q_0 , at any stage if TM finds the blank symbol then TM has found a palindrome of even length. Otherwise, it notes the symbol being read and attempts to match it with last non-blank symbol on the tape. If the symbol is **a**, the TM replaces it by # goes to state q_1 , in which it skips all a's and b's and on #, the TM from q_1 will go to q_3 to find a matching a in last non-blank symbol position. If a is found, TM goes to q_5 replace a by #. However, if b is found then TM has no more indicating the string is not a palindrome. However, if in state q_2 only #'s are found, then it indicates that the previous 'a' was the middle most symbol of the given string indicating palindrome of odd length.

Similar is the case when b is found in state q_0 , except that the next state is q_2 in this case and roles of a's and b's are interchanged in the above argument.

- (ii) The fact of a string not being a palindrome is indicated by the TM when in state q_3 the symbol b is found or in state q_4 the symbol a is found. The initial configuration is q_0babb .

The required computations are:

- (i) $q_0babb \# \rightarrow q_2babb \# \rightarrow \#aq_2bb \# \rightarrow \#abbq_2\# \rightarrow \#abq_4b \rightarrow \#aq_5b\# \rightarrow \#q_5ab \rightarrow q_5\#ab \rightarrow \#q_0ab \rightarrow \#\#q_1b \rightarrow \#bq_1\# \rightarrow \#\#q_3b$,
As there is no move in state q_3 on b, therefore, string is not accepted.
- (ii) The initial configuration is q_0bbb . Consider the computation:
 $q_0bbb \rightarrow \#q_2b \rightarrow \#bq_2 \rightarrow \#q_4b\# \rightarrow q_5\#\#\# \rightarrow q_0\# \rightarrow h\#$
(We may drop #'s in the rightmost positions).
- (iii) The initial configuration is q_0bab . Consider the computation:
 $q_0bab \rightarrow \#q_2ab \rightarrow \#aq_4b \rightarrow \#q_5a\# \rightarrow q_5\#\# \rightarrow \#q_0\# \rightarrow h\#$
(Note \rightarrow^* denotes sequence of any finite number of \rightarrow).

Exercise6: The transition diagram of the required TM is as shown below.

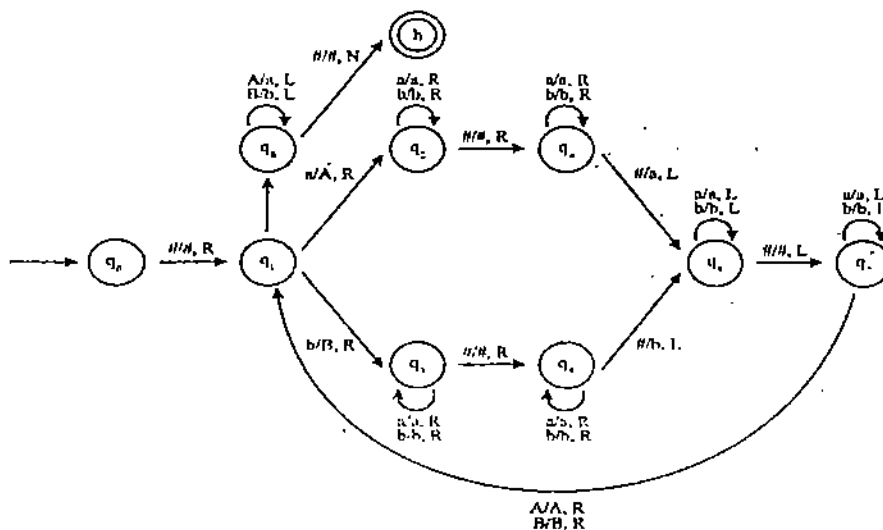


Fig. 1.10.6

The required TM $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ with
 $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, h\}$
 $\Sigma = \{a, b\}$ and $\Gamma = \{a, b, \#\}$.

The next move function is given by the transition diagram above.
 In the solution of the problem, we can deviate slightly from our convention of placing the input string on the left-most part of the tape. In this case, we place # in the leftmost cell of the tape followed by the input string. Therefore, in the beginning in the initial state q_0 , the TM is scanning # in stead of the first symbol of the input. Before we outline the functioning of the proposed TM let us know that for the input string aab is placed on the tape as

#	a	A	b	#	#	***
---	---	---	---	---	---	-----

and for the input, output on the tape is of the form

#	a	a	b	#	a	a	b	#	#	***
---	---	---	---	---	---	---	---	---	---	-----

Outline of the functioning of the proposed TM

The TM in state q_1 notes the leftmost a or b, replaces it by A or B respectively and copies it in the next available # (the first # on the right is left as marker and is not taken as available). If the symbol in the state q_1 is a, then TM while skipping symbols passes through state q_2 and reaches q_4 . However, if the symbol in state q_1 is b, then TM while skipping symbols passes through state q_3 and reaches state q_5 . Then TM copies the symbol and reaches the state q_6 . Next, TM starts its leftward journey skipping over a's, b's, A's, B's and # and meets A or B in q_7 . At this stage, TM goes to state q_1 . Then repeats the whole process until the whole string is copied in the second part of the tape.

But, in this process original string of a's and b's is converted to a string of A's and B's. At this stage TM goes from q_1 to state q_8 to replace each A by a and each B by b. This completes the task.

The Computation of the TM on input aab

The initial configuration is $q_0\#abb$. Therefore, the computation is

$q_0\#abb$		$\#q_1abb$		$\#Aq_2ab$
		$\#Aaq_2b$		$\#Aabq_2\#$
		$\#Aab\#q_4$		$\#Aabq_5\#a$
		$\#Aabq_6\#a$		
		$\#Aq_6ab\#a$		$\#q_6Aab\#a$
		$\#Aqab\#a$		

(At this point whole process is repeat and, therefore, we use \vdots , representing a finite number of \vdots)

\vdots		$\#AAq_0b\#aa$
\vdots		$\#AABq_0b\#aab$

At this stage TM enters state q_7 .

		$\#AAq_7B\#aab$
		$\#Aq_7Ab\#aab$
		$\#q_7Aab\#aab$
		$q_7\#Aab\#aab$
		$h\#aab\#aab$

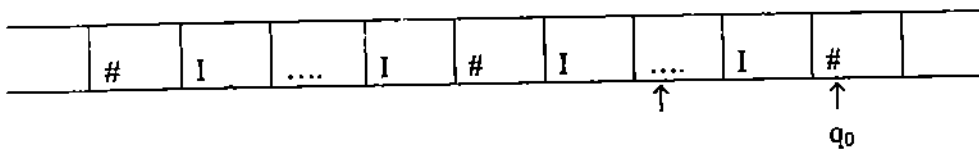
Exercise7 : In respect of the design of the TM $(Q, \Sigma, \Gamma, \delta, q_0, h)$, where $\Sigma = \{1\}$, $\Gamma = \{1, \#\}$ where we made the following observations:

Observation1: General form of the tape is

	#	1	1	#	1	..	1	#
--	---	---	------	---	---	---	----	---	---

There are three significant positions of #, which need to be distinguished viz right-most # on left of I's, middle #, middle # and left-most # on the right of I's. Therefore, there should be change of state on visiting each of these positions of #.

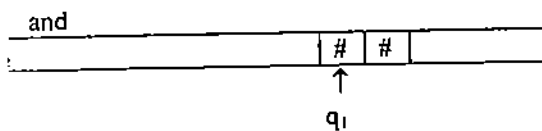
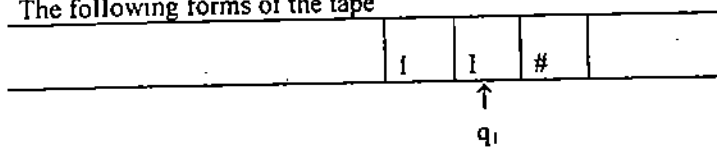
Observation2: Initial configuration is



and as observed above

$$\delta(q_0, \#) = (q_1, \#, L)$$

The following forms of the tape



guide us to moves

$$\delta(q_1, I) = (q_2, \#, L)$$

change of state is essential else other I's will also be converted to #'s,

$$\delta(q_1, \#) = (\text{halt}, \#, N)$$

Observations3: The moves are guided by principle that convert the left-most I to # on the right side the corresponding right-most I to # on the left-side

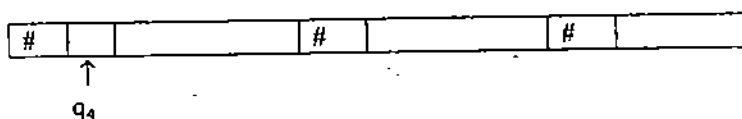
$$\delta(q_2, I) = (q_2, I, L)$$

$$\delta(q_2, \#) = (q_3, \#, L)$$

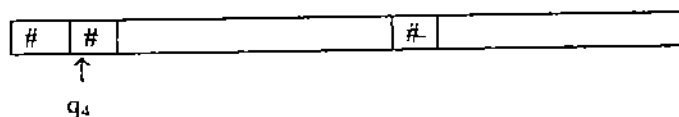
$$\delta(q_3, I) = (q_3, I, L)$$

$$\delta(q_3, \#) = (q_4, \#, R)$$

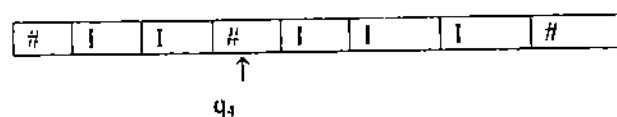
(We have reached the right-most # on the left of all I's as shown below)



If we have configuration of the form



then it must have resulted from initial configuration in which $m < n$ represented by say



Therefore, we must now enter a state say q_7 which skips all I's on the right and then halts

Therefore

$$\begin{aligned} \delta(q_4, \#) &= (q_7, \#, R) \\ \delta(q_7, I) &= (q_7, I, R) \\ \delta(q_7, \#) &= (\text{halt}, \#, N) \end{aligned}$$

Next, we consider $\delta(q_4, I)$

$$\delta(q_4, I) = (q_5, \#, R)$$

(state must be changed otherwise, all I's will be changed to #s)

$$\delta(q_5, I) = (q_5, I, R)$$

$$\delta(q_5, \#) = (q_6, \#, R)$$

(the middle # is being crossed while moving from left to right)

$$\delta(q_6, I) = (q_6, I, R)$$

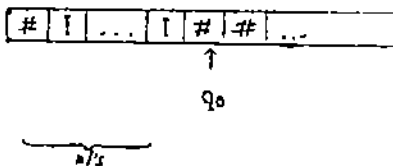
$$\delta(q_6, \#) = (q_0, \#, N)$$

(the left-most # on right side is scanned in q_6 to reach q_0 so that whole process may be repeated again.)

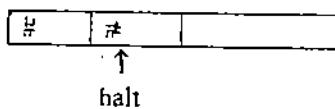
Summarizing the above moves the transition table for δ function is given by

	I	#
q_0		$(q_1, \#, L)$
q_1	$(q_2, \#, L)$	$(\text{halt}, \#, L)$
q_2	(q_2, I, L)	$(q_2, \#, L)$
q_3	(q_2, I, L)	$(q_4, \#, L)$
q_4	$(q_5, \#, R)$	$(q_7, \#, R)$
q_5	(q_5, I, R)	$(q_6, \#, R)$
q_6	(q_6, I, R)	$(q_6, \#, R)$
q_7	(q_7, I, R)	$(\text{halt}, \#, N)$
Halt	-	-

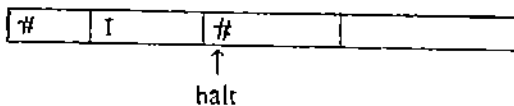
Exercise8: By our representation conventions, the initial configuration is as follows



If n is even, then $f(n) = 0$ which further is represented by final configuration



If n is odd, then $f(x) = 1$ which is represented by $f(n) = 1$ which is represented by a final configuration of the form



The strategy of reaching from initial configuration to a final configuration is that after scanning even number of I's we enter state q_2 and after scanning odd number of I's, we enter state q_1 and then take appropriate action, leading to the following (partial) definition of transition function δ :

- $\delta(q_0, \#) = (q_2, \#, L)$
- $\delta(q_2, I) = (q_1, \#, L)$
- $\delta(q_2, \#) = (\text{halt}, \#, N)$
- $\delta(q_1, I) = (q_2, \#, L)$
- $\delta(q_1, \#) = (q_3, \#, R)$
- $\delta(q_3, \#) = (\text{halt}, I, R)$

For the transition

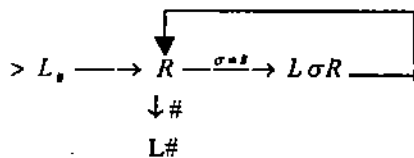
$\delta(q_i, a_i) = (q_j, a_i, m)$, the sequence of actions is as follows: First a_i is written in the current cell so far containing a_i . Then movement of tape head is made to left, to right or 'no move' respectively according as the value of m is L, R or N. Finally the state of the control changes to q_j .

The transition function δ for the above computation is

δ	#	I
q_0	$(q_2, \#, L)$	$(q_1, \#, L)$
q_1	$(q_3, \#, R)$	$(q_2, \#, L)$
q_2	$(\text{halt}, \#, N)$	$(q_1, \#, L)$
q_3	(halt, I, R)	-
halt	-	-

The students are advised to make transition diagram of the (partial) function defined by the above table.

Exercise 9: The desired machine S_L is given by



Exercise 10: Hint: The machine CS_L obtained by composing the earlier designed two machines C and S_L is the required machine.

Exercise 11: The proposed design is broken up into a number of the following steps:

Step I: is to mark the left end of the tape by writing a non-blank character say d in the left-most cell after shifting the given string to the Right.

Thus we apply S_R which transforms the tape configuration.

$\# \omega \#$ with $\omega \in \Sigma^*$

to the configuration

$\# \# \omega \omega \#$

And then we write d in the left most cell so that tape configuration becomes

$d \# \omega \#$

And the component TM for Step 1 is given by $S_R L \# L d$.

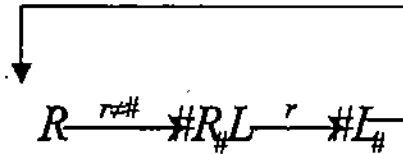
Step II: In order to move to the left-most non-blank symbol of the original string, apply R to reach the $\#$ which is to the left of the left-most non-blank symbol.

Step III: The following moves are repeatedly applied:

- (i) Apply R to move to the left-most non-blank. The current symbol is read as r and then replaced by $\#$. Then this r is attempted to be matched with the right-most

non-blank symbol. The right-most non-blank symbol is reached by applying R_rL . Thus total machine component of Step III (i) is given by $\#R_rL$.

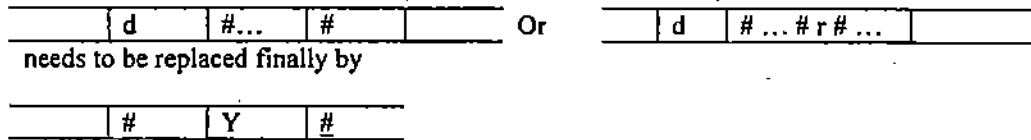
- (ii) *At this stage one of the possibilities is that the symbol currently being scanned is same as r.*
 In this case the symbol is replaced by # and then the #, if any, to the left of non-blank part is reached through L_r .
 Whole process is repeated.
 The TM component of the part discussed so far, Step III is of the form



At some stage, either of the following three cases happen.

- (a) the tape contains string of #'s only.
- (b) only one non-# symbol is left on the tape.
- (c) right-most non-# symbol does not match r.

Out of these three cases, in the first two cases, the given string is a palindrome and hence the tape configuration.



In the third case (c) above, at some stage the tape may be of the form
 $d \# \# \# \dots \# \# \dots \dots \dots b \# \# \dots \#$

In this case first all non-blanks need to be replaced by blanks and then final Tape configuration should be

$\# N \#$

First we discuss the cases when the string is a palindrome and hence we need to replace the configuration

$d \# \dots \#$ OR $d \# \dots \# r \# \dots \#$

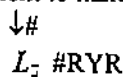
(In the configurations Head is scanning # or) by the configuration

$\# y \#$

In such cases at some stage, the current symbol is # or r, some non-blank symbol .

Let us first consider the case when head scans # (i.e. case of even length palinidrome)

We move to left-most symbol d through L_r , replace d by # then, move to the Right to write Y in the cell under the Head and finally move to the Right. Thus the TM component to handle this part is given by:



after R component of the component TM of Stage II.

Next we discuss the case when the initially given string is a 'palindrome of the form.
 $\# a b c b a \#$

for which, after a number of moves, the following configuration is reached:

$$d \# \# \# c \# \# \#$$

Then c is replaced by #. While executing $R \xrightarrow{r\#} \# R_\#$ part of the following component of the TM of Step III

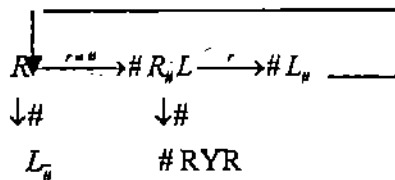
$$R \xrightarrow{r\#} \# R_\# \quad L \xrightarrow{r} \# L_\#$$

leads to the configuration $d \# \# \# \# \# \#$ by replacing c by # and moving to the next # on the right. Next executing L of $L \xrightarrow{r} \# L_\#$ takes us back to the configurations $d \# \# \# \# \# \#$, where in the case of even palindromes or for all states except last for odd palindromes, we expect under the Head at this stage, the previously noted symbol. But in this case it does not happen, because # is present in stead of the expect symbol c.

Therefore the part $\xrightarrow{r} \# L_\#$ is not executed.

Therefore, there is an accepting branch # from L.. Afterward, actions are similar as in the case of even palindrome discussed above.

Combining the two cases we get the following component of the TM which correspond to the two cases of acceptance as Palindrome of the given string:



Case (iii) When the given string is not a palindrome and we have already reached a stage where the corresponding positions do not have the same letter e.g.

$$d \# a \# b \# c \# c \# c \# a \#$$

in which after having executed

$$R \xrightarrow{r\#} \# R_\# \quad L \xrightarrow{r\#} \# L_\#$$

once completely and only upto $\xrightarrow{r\#} \# R_\# L$ in the second round we find a 'c' (instead of expected 'b')

∴ at the stage to the component $R \xrightarrow{r\#} \# R_\# L$ we add

another arc $t \neq r$ or #

(in addition to the arc \xrightarrow{r} when the pair of letters in corresponding positions match) as shown in the lower right part of the next diagram.

Action-to-be defined once a non-palindrome is recognized: Replace all non-blanks by blanks so that the tape assumes the configuration.

$$d \# \dots \# \#$$

which finally through a series of actions assumes the form

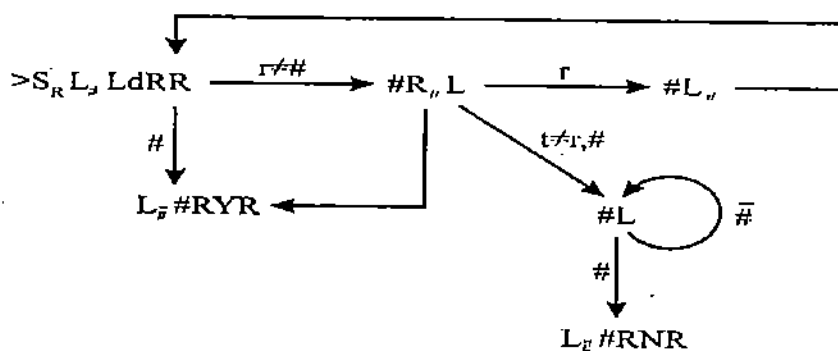
$$\# N \#$$

Coming back to the latest non-accepting configuration, the Head is scanning a non-blank (in our example 'c'), we replace it by # and move to the next non-blank (if any) on the left-side, i.e., apply $L_\#$. Thus application of $L_\#$ is repeated as long as there

are non-blanks available on the tape. Also, as all non-blanks are continuous, therefore, in stead of $\# L_{\bar{r}}$, we may take only $\# L$. As all non-blanks are continuous, therefore, we reach $\#$ only when the configuration is of the form $d \# \#$. This configuration is to be replaced by

$$\# N \#$$

The sequence of actions required for the final configuration is $L_{\bar{r}} \# R N R$. After combining all the above sub-machines, we get



- Fig. 1.10.7

1.11 FURTHER READINGS

1. H.R. Lewis & C.H.Papadimitriou: Elements of the Theory of computation, PHI, (1981)
2. J.E. Hopcroft, R.Motwani & J.D.Ullman: Introduction to Automata Theory, Languages, and Computation (II Ed.) Pearson Education Asia (2001)
3. J.E. Hopcroft and J.D. Ullman: Introduction to Automata Theory, Language, and Computation, Narosa Publishing House (1987)
4. J.C. Martin: Introduction to Languages and Theory of Computation, Tata-Mc Graw-Hill (1997).

UNIT 2 TURING MACHINE – MISCELLANY

Structure	Page Nos.
2.0 Introduction	55
2.1 Objectives	56
2.2 Extensions-cum-Equivalents of Turing Machine	56
2.3 Universal Turing Machine (UTM)	68
2.4 Languages Accepted/Decided by TM	72
2.5 The Diagonal Language and the Universal Language	78
2.6 Chomsky Hierarchy	84
2.7 Summary	88
2.8 Solutions Answers	88
2.9 Further Readings	91

2.0 INTRODUCTION

For the time being, let us concentrate on the nitty-gritty of other, possibly easier, ways of designing TMs and other related issues, and leave the issue of self-reference for some later units.

The essence of the discipline of *Theory of Computation* is to characterize the *phenomenon of computation* in terms of formal/mathematical concepts like set, relation, function, etc. For this purpose, the discipline incorporates study of a number of approaches to, and models and principles of, computation. Three approaches to computation included in the curriculum are:

- (i) Automata
- (ii) grammatical and
- (iii) recursive function.

Various approaches to computation are equivalent in the sense that to each model of computation obtained through one approach, there is a (computationally) equivalent model of computation through another approach.

We initiated our studies with Finite Automata and Regular Grammars and established equivalence of these models. However, these models are found inadequate to capture the notion of computation, in the sense that even a simple language like $\{x^n y^n : n \in \mathbb{N}\}$ cannot be captured/computed by either of these models. Then, we studied more powerful models viz. Pushdown Automata and Context-Free Grammars and established equivalence between the models. Again, these models are found inadequate.

In the previous unit, we introduced still more powerful model of computation viz Turing Machine (TM) and mentioned *the important fact that that TM model is conjectured to be the ultimate (formal) model of computation.*

In this unit, we discuss a number of important issues about TM. First of all, we mention a number of extensions of the standard TM introduced in the previous unit. These extensions, though apparently are expected to provide more powerful models, yet give only models, each one of which is equivalent to standard TM. The fact of equivalence of various extensions of TM support the conjecture mentioned above. *The proofs of equivalences are beyond the scope of the course.*

Tortoise: Oh, how clever. I wonder why I never thought of that myself. Now tell me: is the following sentence self-referential? "Is Composed of Five words." "Is Composed of Five Words."

Achilles: Hmm... I can't quite tell. The sentence which you just gave is not really about itself, but rather about the phrase "is composed of five words". Though, of course, that phrase is part of the sentence

Tortoise: So the sentence refers to some part of itself — so what?

Achilles: Well, wouldn't that qualify as self-reference, too?

Tortoise: In my opinion, that is still a far cry from true self-reference. But don't worry too much about these tricky matters. You'll have ample time to think about them in the future.

Hofstadter**

** Godel, Escher, Bach: An Eternal Golden Braid By Douglas R. Hofstadter, Penguin Books (1979)

Next, we discuss *Universal Turing Machine (UTM)*, an equivalent of *general-purpose computer*. The significance of the study of UTM lies in the facts:

- (i) A single General Purpose Computer can be used to solve any problem, if at all the problem is solvable by some computational method.
- (ii) In order to solve a problem by TM model, *unlike general purpose computer*, we are required to *construct a new TM for each new problem*.

Thus, a single UTM can be used to solve by TM models *any* solvable problem. Next, we introduce languages associated with TM and discuss briefly properties of these languages.

Though, some of the books that have appeared in the recent past in the discipline, do not talk of Chomsky* Hierarchy of languages; we, for the sake of exhibiting complete parallel between the automata and grammar approaches, just mention Chomsky Hierarchy and define grammar models of various types of languages discussed under Chomsky Hierarchy and mention equivalences of these languages to appropriate automata

2.1 OBJECTIVES

After going through this unit, you will be able:

- to discuss various extensions of standard Turing Machine;
- to tell that each of these extensions of TM, is just computationally equivalent and, is not properly more powerful than standard TM;
- to describe the structure of Universal Turing Machine (UTM);
- to explain how UTM can be used as a general purpose computer;
- to state and prove some of the properties of Turing Acceptable and Turing Decidable languages; and
- to define phrase-structure grammar and to tell that phrase-structure grammar model is equivalent to TM model.

2.2 EXTENSIONS-CUM-EQUIVALENTS OF TURING MACHINE

The Turing Machine, as defined in the previous unit, will be referred to as *standard Turing Machine*. In the standard Turing Machine, the tape is *semi-infinite* and is bounded on the left-end, however, the tape is unbounded on the right side. In this section we consider some extensions of the standard TM.

The extensions of Turing Machine considered are:

- (i) The tape may be allowed to be *infinite in both* the directions
- (ii) There may be *more than one Head* scanning various cells of the tape. Two or more Heads may simultaneously read the same cell or may attempt to write in the same cell.
- (iii) There may be *several Tapes* instead of one only, each Tape having its own independent Head.
- (iv) The Tape may be *k-dimensional*, $k \geq 2$, instead of only one-dimensional.

- (v) For a given pair of current state and symbol under the Head, *in stead of* at most one possible move, there may be any finite, possibly zero, number, of next moves (*This model is called Non-Deterministic Turing Machine.*).

Remark 2.2.1

In all the above-mentioned extensions, it is invariably assumed that *only finitely many* cells contain non-blank symbols. All other cells are blanks.

Remark 2.2.2

Each of the above-mentioned extensions, being a generalization of the standard Turing Machine, *may appear to yield a strictly more powerful model of computation* through automata approach, *yet it has been proved* that each of these models is just equivalent to and *not strictly more powerful* than the standard TM model of computation.

It has been already mentioned in one of the previous units that it is conjectured that (standard) TM is ultimate model of computation.

Remark 2.2.3

Like the standard TM, each of the extensions of TM enumerated above, is formally defined as, or some variation of, a sextuple of the form $(Q, \Sigma, \Gamma, \delta, q_0, h)$, where Q ; Σ , Γ , q_0 and h stand, as in standard TM, for respectively set of states, set of input symbols, set of Tape symbols, initial state and halt state.

However, the extensions are distinguished from each other and from the standard TM through different definitions of next-move relation δ and of configurations for each of the extension. Therefore, in the following, most of the time, we discuss the extensions *only in terms of definitions of δ and of configuration.*

2.2.1 Extension (i):

Two-way (infinite tape) Turing Machine

Like standard TM, in this case also, the next-move is given by δ as a *partial function* from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R, N\}$

The following three points need to be noted in respect of configurations of Two-way Turing Machine:

(i) Configuration/Instantaneous Description:

In standard TM, if there are a number of left-most positions which contain blanks, then those are included in the configuration, e.g. if the one-way configuration Tape is of the form

a b # c d e f # #..#
 ↑
 q_2

then the configuration in the *standard* TM is written as:

$(q_2, \# \# a b \# c d e f)$,

where we neglect all the continuous sequences of right-hand blanks.

However, in the Two-way infinite Tape TM, both left-hand and right-hand parts of the tape are symmetrical in the sense that there is an infinite continuous sequence of blanks on each of the right-hand and left-hand of the sequence of non-blanks.

Therefore, in the case of *two-way infinite Tape*, if the above string is on the tape then it will be in the form

Where S_i is the symbol to be written in the cell under H_i , the i th Head and M_i denotes the movement of H_i , where the movement may be L, R or N and further L denotes movement to the left, R denotes movement to the right of the current cell and N denotes 'no movement of the Head'

Two Special cases of the δ function defined above, need to be considered:

- (i) What should be written in the current cell when both Heads are scanning the same cell at a particular time and the next moves $(S_1, M_1), (S_2, M_2)$ for the two Heads, are such that $S_1 \neq S_2$ (i.e. symbol to be written in current cell by $H_1 \neq$ symbol to be written in current cell by H_2)?
In such a situation, a general rule may be defined, say, as 'whatever is to be done by H_1 will take precedence over whatever is to be done by H_2 '.
- (ii) **The Hanging configuration:** For two-Head One-way Tape, a configuration shall be called *Hanging* if
 $\delta(q, \text{symbol under } H_1, \text{symbol under } H_2)$
 $= (p, (S_1, M_1), (S_2, M_2))$
 is such that either
 - (a) Symbol under H_1 is in the left-most cell and M_1 is L, i.e., movement of H_1 is to be to the left. **OR**
 - (b) Symbol under H_2 is in the left-most cell and M_2 is L, i.e., movement of H_2 is to be to the left.

Other concepts and issues in respect of Two-Head One-way Tape may be handled on the similar lines. *The above discussion can be further be extended easily to the case when number of Heads is more than two.*
Again, as mentioned earlier, the power of the TM is not enhanced by the use of extra Heads.

2.2.3 Extension (iii)

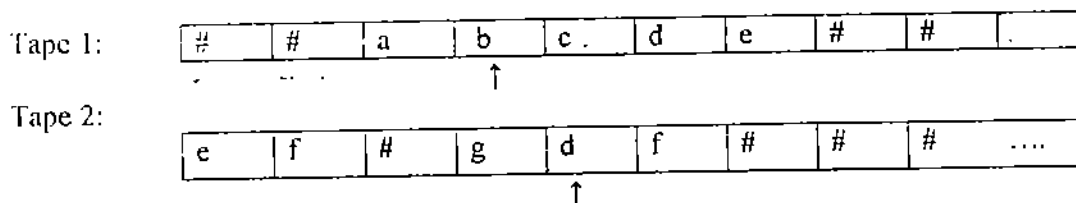
Multi-Tape Turing Machine:

In stead of one Tape, we may have more than one tapes, each tape having its own independent Head. To begin with, we may take each of the tape as *one-way infinite tape, bounded on the left*.

Again to facilitate the discussion, we initially consider the case of *only two tapes*:

Configuration/Instantaneous Description:

We explain the concept of *configuration for Turing Machine with two Tapes* with an example. Let the *contents of the tapes and positions of the Heads* be as follows:



and the state of the Turing Machine be q .
Then the *configuration* may be denoted by
 $(q, (\# \# a \underline{b} c d e), (e f \# g \underline{d} f))$

(inner pairs of parentheses are used only to enhance readability, not required otherwise)

The next Move function δ may be defined as

$$\delta((q, T_1, T_2)) = (p, (S_1, M_1), (S_2, M_2))$$

where q denotes the current state, T_i denotes the symbol of the i th tape currently being scanned by its Head. The symbol p denotes the next state; S_i denotes the symbol to be written in the current cell of the i th Tape in place of T_i . $M_i \in \{L, R, N\}$ denotes the movement of the Head on i th Tape.

Hanging Configuration in the case of Two-Tape, each Tape being one-way infinite

The TM will be said to be in Hanging Configuration if there is a next move given by $\delta(q, T_1, T_2) = (p, (S_1, M_1), (S_2, M_2))$, where p, q, T_i, S_i, M_i , are the notations explained above, with either

- (i) T_1 being in the left-most cell of Tape 1 and M_1 being 'Movement to Left', or
- (ii) T_2 being in the left-most cell of Tape 2 and M_2 being 'Movement to Left'.

The discussion can be further extended on the similar lines to k Tape Turing Machine, where $k > 2$.

The concept of k -Tape, $k \geq 2$, with each Tape being semi-infinite, can be further extended when the tapes are allowed to be Two-way infinite. The notions for configuration and Move function for such machines can be easily defined.

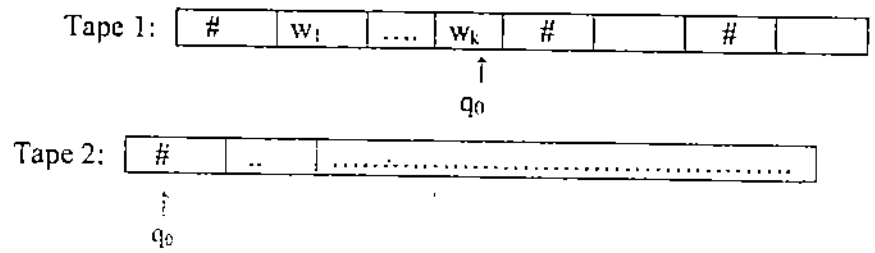
A very important application of the 3-tape Turing Machine model, which we are going to discuss in Section 2.3, is in the design of universal Turing Machine, a sort of a general-purpose computer.

The design of k -tape Turing Machines for some of the functions like copying, reversing, for verifying whether a string is a palindrome or not etc, can be much more easily carried out as compared to the design of the corresponding standard Turing Machines.

Example: 2.2.3.1

Construct a 2-Tape Turing Machine, which returns $\# \omega \#$ for given input $\# \omega \#$.

Solution: Let the input be placed on Tape 1 and Tape 2 may contain all blanks, with the Head of Tape 2 being on the left-most $\#$ so that the initial configuration is as follows:



Step1: Move the Head of Tape 1 containing the input towards the left most cell through the following moves.

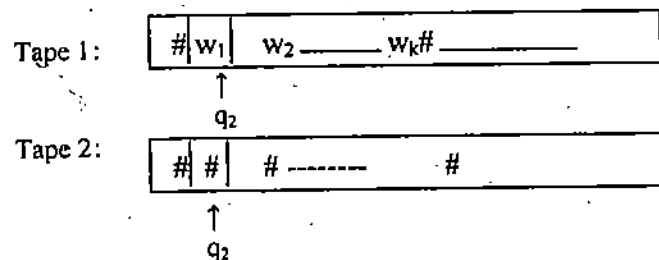
$$\delta(q_0, \#, \#) = (q_1, (\#, L), (\#, N))$$

$$\delta(q_1, \bar{\#}, \#) = (q_1, (\bar{\#}, L), (\#, N))$$

$$\delta(q_1, \#, \#) = (q_2, (\#, R), (\#, R))$$

where $\bar{\#}$ denotes the same non-blank symbol throughout an equation.

After these moves, the configuration is as follows:



where $w_i \neq \#$ for $i=1, 2, \dots, k$

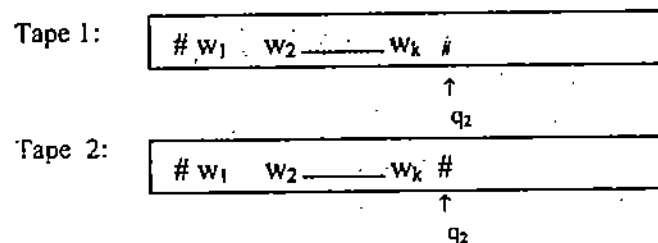
Step 2: Next, we copy the contents of Tape 1 to Tape 2 through

$$\delta(q_2, \bar{\#}, \#) = (q_2', (\bar{\#}, R), (\bar{\#}, R)),$$

where $\bar{\#}$ denotes the same non-blank symbol throughout an equation.

In other words through these k moves, non-blank contents of Tape 1 are copied in the corresponding cells of tape 2.

After k times executions of the above move, the configuration becomes



Step 3: At this stage we intend to move the Head of Tape 2 to the left-most # without moving the Head of Tape 1

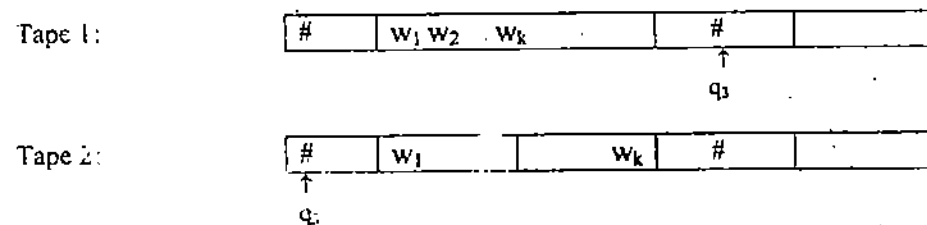
we introduce the moves:

$$\delta(q_2, \#, \#) = \delta(q_3, (\#, N), (\#, L))$$

and

$$\delta(q_3, \#, \bar{\#}) = (q_3, (\#, N), (\bar{\#}, L))$$

At the end of k moves the configuration becomes



At this stage, when Head of Tape 2 is also scanning a #, we may enter a new state q_4 , in which Head of Tape 1 does not move but Head of Tape 2 moves right so that

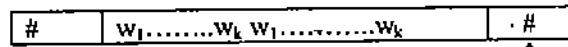
$$\delta(q_3, \#, \#) = (q_4, (\#, N), (\#, R)) \dots (*)$$

In state q_4 , each non-# symbol of Tape 2 is copied in the current cell of Tape 1, and then content of the current cell of Tape 2 is converted to # and both Heads move to the Right i.e.,

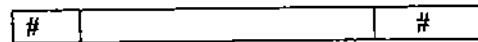
$$\delta(q_4, \#, \bar{\#}) = (q_4, (\bar{\#}, R), (\#, R))$$

Step 4: Finally the configuration with state q_4 is

Tape 1



Tape 2



$$\therefore \delta(q_4, \#, \#) = (\text{Halt}, \#, \#)$$

At this stage Tape 1 contains the required output.

Ex.1) Construct Two-Tape Turing Machines for each of the following:

- (i) Convert the input # w # into # w # w #
- (ii) Convert the input # w # into # w w^R #
- (iii) Convert the input # w # into # w # w^R #

where if $w = w_1 w_2 \dots w_{k-1} w_k$
 then $w^R = w_k w_{k-1} \dots w_2 w_1$

Remark 2.2.3.2:

Again, it has been proved that the power of the *standard Turing Machine* is the same as that of a *Turing Machine with any finite number of Tapes*.

Remark 2.2.3.3:

The *k-Tape version of a Turing Machine*, with each tape being only one-way can be further extended to a *k Tape Turing Machine with each Tape being Two way infinite*. It may again be noted that *even with this extension the computing power is the same* as is achievable with standard TM.

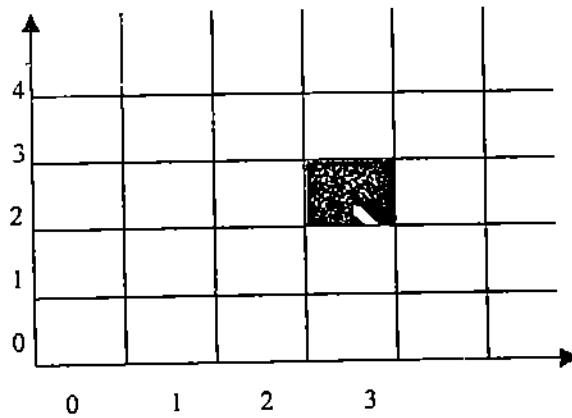
Next, let us consider

2.2.4 Extension (iv):

k-Dimensional Turing Machine:

Again to facilitate the understanding of the basic ideas involved, let us discuss initially only *Two-Dimensional Turing Machine*. Then these ideas can be easily generalized to *k-dimensional case*, where $k > 2$.

In the case of two-dimensional tape as shown below, we assume that the tape is bounded on the left and the bottom.



Each cell is given an address say (i_1, i_2) where i_1 is the row-number of the cell and i_2 is the column number of the cell. For example, the shaded cell in the above diagram has address $(2,3)$.

Introductory Remarks in context of the Instantaneous Description (ID) or configuration:

A configuration of a two-dimensional TM at a particular time may be described in terms of finitely many of the triplets of the form, (i_1, i_2, c) where for each such triplet, (i_1, i_2) is the address of a cell and c denotes the contents of the cell. Only these cells are included in an ID, for which c , the contents, are non-blank symbols.

In the configuration or ID, order of the cells which are included in an ID, Row-Major Ordering is to be followed, i.e., first all the elements in the row with least index are included in the ID, followed by the elements of the row with next least index and so on. Within cells of each row, the cell with non-# contents and having least column number is included first followed by the non-# cell with next least column number and so on.

For example, if we have the following triplets in the ID
 $(2,5, c), (0,2,d), (4,3, f), (3,5,g), (0,3,h)$,
 then the order of the triplets in the ID will be
 $(0,2,d), (0,3,h), (2,5,c), (3,5,g), (4,3,f)$

After these introductory remarks, we define configuration and the move function δ etc.

Configuration: Let $q \in Q, c_k \in \Gamma \sim \{\#\}$.

i.e. c_k is a non-blank Tape symbol.

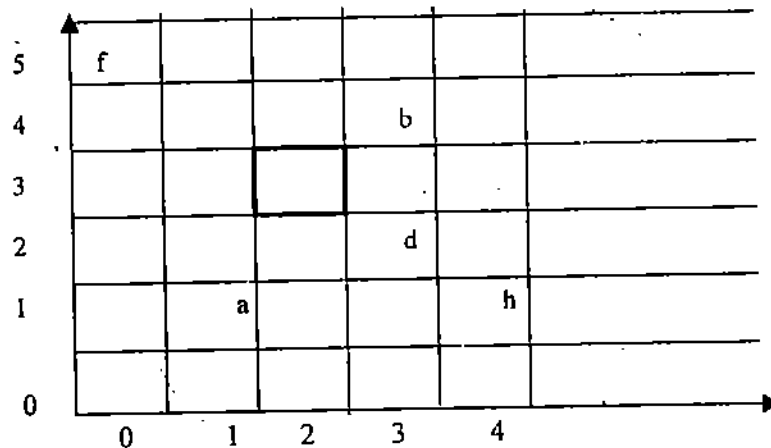
Then a configuration at a particular instant is denoted by
 $(q, (H_1, H_2) ((i_1, i_2, c_{i_1, i_2}), (j_1, j_2, c_{j_1, j_2}), \dots, (k_1, k_2, c_{k_1, k_2}) \dots \dots \dots))$,
 where each of $c_{i_1, i_2}, c_{j_1, j_2}, \dots \dots \dots$ is non-blank and these are the only non-blanks on the tape.

Also, (H_1, H_2) denotes the location of the cell currently being scanned, i.e. the cell under the Head.

Further, (i_1, i_2) precedes (j_1, j_2) and (j_1, j_2) precedes (k_1, k_2) in the row-major ordering, if
 $i_1 < j_1 < k_1$
 and if $i_1 = j_1$, then $i_2 < j_2$
 or if $j_1 = k_1$, then $j_2 < k_2$ etc.

Example 2.2.4.1:

Suppose at a particular instant the contents of a Two-Dimensional Tape are as given below and the state at that instant is q_3 and the cell being scanned is (3,2).



Then the configuration / ID is given by $(q_3, (3,2), (1,1,a), (1,4,h), (2,3,d), (4,3,b), (5,4,f))$

The Next-Move function δ : maps an element of $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R, U, D, N\}$, where L, R, U and D denote respectively 'Move Left', 'Move Right', 'Move Up' and 'Move Down', and 'N' denotes 'No Move'. For example, $\delta(q_2, c) = (q_3, d, R)$

means the contents viz c of the cell (i_1, i_2) currently being scanned, are replaced by d and the Head moves to the cell with address $(i_1, i_2 + 1)$ if the address of the scanned cell was (i_1, i_2) .

The following cases need special attention:

The cases are discussed only in respect of inclusion or exclusion of triplets and not about movement of the Head.

Let $\delta(q_2, c) = (d, n)$.

Case (i) if $c = \#$ then $(i_1, i_2, \#)$ does not occur among the triplets of the configuration before the move. However if $d \neq \#$ then (i_1, i_2, d) will be added to the set of triplets in the configuration.

Case (ii) if $c \neq \#$ but $d = \#$ then (i_1, i_2, c) occurs as a triplet in the configuration before the move, but this triplet is dropped from the new configuration arising out of $\delta(q_2, c) = (d, n)$.

Case (iii) When $c = d = \#$
In this case, there is no change in the set of triplets in the configuration $\delta(q_2, c) = (d, n)$.

Case (iv) When $c \neq \#$, and $d \neq \#$, then the triplets (i_1, i_2, d) replaces the triplet (i_1, i_2, c) in the set of all triplets in the previous configuration to get the new configuration $\delta(q_2, c) = (d, n)$.

Again, it has been proved that the computing power of the above-mentioned model of TM remains the same as that of the standard TM.

Next, we come to the most important extension of the TM, viz

2.2.5 Extension v:

Non-Deterministic Turing Machine. (NDTM)

An NDTM is like the standard TM with the difference as described below. In Standard TM, to each pair of the current state (except the halt state) and the symbol being scanned, there is a unique triplet comprising of the next state, unique action in terms of writing a symbol in the cell being scanned and the motion, if any, to the right or left. However, in the case NDTM, to each pair (q, s) with q as current state and s as symbol being scanned, there may be a finite set of the triplets $\{(q_i, s_i, m_i) : i = 1, 2, \dots\}$ of possible next moves. This set of triplets may be empty, i.e. for some particular (q, s) the TM may not have any next move. Or alternatively the set $\{(q_i, s_i, m_i)\}$ may have more than one triplet, meaning thereby that the NDTM in the state q and scanning symbols s , has the alternatives for next move to choose from the set $\{(q_i, s_i, m_i)\}$ of next moves.

It can be easily seen that standard TM is a special case of the NDTM in which for each (q, s) the set $\{(q_i, s_i)\}$ of next moves is a singleton set or empty.

In order to define formally the concept of Non-Deterministic TM (NDTM), and a configuration in NDTM etc, we assume that the tape is one-way infinite.

For the extensions of the standard TM, discussed so far, we did not state the full formal definition of each of the extension. We only discussed the definition only relative to the standard TM. Mainly we discussed configurations and partial move function δ for each of the extensions. However, in view of the significant though small, difference in the behaviour of an NDTMs, we provide below full formal definition of NDTM.

Remark 2.2.5.1:

An important point about the definition of NDTM needs to be highlighted. By the definition of δ which maps an element of (q, x) of $Q \times \Gamma$ to a set $\{(q_i, x_i, M_i)\}$ means that each element (q, x) of $Q \times \Gamma$ has the potential of leading to more than one configurations. In other words, there are various possible routes to a final configuration from one configuration. However, during one computation only one of these possible values (q_i, x_i, M_i) will be associated with (q, x) through δ . But we can not tell in advance which one out of the ordered triples from the set $\{(q_i, x_i, M_i)\}$

This is why the adjective Non-Deterministic is used for this version of the T.M.

Remark 2.2.5.2:

The set $\{(q_i, x_i, M_i)\}$ associated with (q, x) under δ , may be empty. This means there is no possible next move for (q, x) , a situation that occurred even in the case of standard TM and other versions discussed so far. This is why δ was called a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R, N\}$.

Remark 2.2.5.3:

In the standard TM and the versions discussed before NDTM, we allowed δ as a partial function to $Q \times \Gamma \times \{L, R, N\}$. In other words, if a value under δ exists for (q, x) then the value has to be unique, i.e. can be determined. Therefore, the earlier versions are prefixed with the adjective *Deterministic*. The Non-Deterministic form of each of the earlier versions can be obtained by making suitable

modifications in the corresponding definitions of δ etc on the lines of modifications suggested in the definition of NDTM from standard TM.

Remark 2.2.5.4:

Proper non-determinism means that at some stage, there are at least two next possible moves. Now, if we engage two different persons or machines to work out further possible moves according to each of these two moves, the two can work **independent** of each other. **This means Non-Determination allows parallel computations.** This characteristic of Non-Determinism, also allows is further computations even if some of the sequences of moves may be locked as there may not be any next moves at some stages.

Definition: An Non-Deterministic Turing Machine

is a sextuple $(Q, \Sigma, \Gamma, \delta, q_0, h)$ where

Q : Set of States

Σ : Set of input symbols

Γ : Set of tape symbols

q_0 : The initial state

h : The halt state and

$\delta: Q \times \Gamma \rightarrow$ Power set of $(Q \times \Gamma \times \{L, R, N\})$

The concept of a configuration is same as in the case of standard TM. But the

concept of 'yields in one step' denoted by \vdash_m , has different meaning. Here one

configuration may yield more than one configurations.

We explain these ideas through a suitable example, which also demonstrates the advantage of the Non-Deterministic Turing Machine over the standard Turing Machine. The advantage is in respect of the relative ease of construction of NDTM.

Remarks 2.2.5.5

Before coming to the example, showing advantage of an NDTM in solving some problems; we need to understand properly the concept of acceptance of a language by an NDTM. First of all, let us recall below what is meant by acceptance of a language L by a standard TM M .

A language L is accepted by a TM M if each string $\alpha \in L$, is acceptable by M .

Further a string α is acceptable M , if starting in the initial state q_0 of M , with α as input on the tape of M , if we are able to reach halt state in a finite number of moves, i.e, if $\alpha = a_1 a_2 \dots a_k \in L$ for $a_i \in \Sigma$, the set of input symbols of M , then

$(q_0, a_1 a_2 \dots a_k) \vdash^* (h, \beta)$

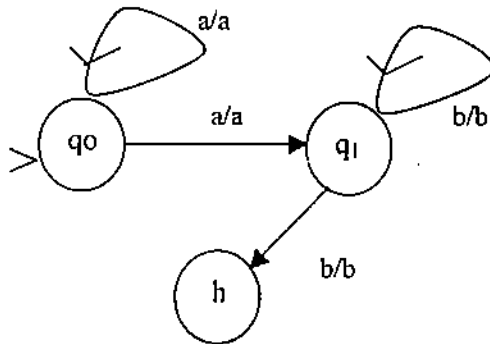
Where β is a string of tape symbol and tape head may be on any cell of the tape. A characteristic feature of the standard TM, in this case, is that if there is to be a sequence of moves from (q_0, α) to a final state, than that sequence might be unique. However in the case of Non-Deterministic machines, the halt state may be reached through any one of various permissible sequences of moves. Therefore in this version a string α over the set of input symbols of an NDTM is acceptable by an NDTM M , if by at least one but by any one of the sequences of moves halt state is reached from (q_0, α) . Now we discuss the example showing advantage of NDTM over standard TM.

Example 2.2.5.6:

Construct an NDTM which accepts the language $\{ a^n b^m : n \geq 1, m \geq 1 \}$, i.e., the language of all strings over $\{a,b\}$, in which there is at least one a and one b and all a's precede all b's.

Solution: The diagrammatic representation of the required NDTM is as given below:

In the proposed NDTM, as the motion of the head is always to the Right except in the Halt state. Therefore, R is not mentioned in the labels in the diagram below:



where the label i/j on an arc denotes that if symbol in the current cell is i then contents of the cell are to be replaced by j .

Formally the proposed NDTM may be defined as

$$M = \{ \{q_0, q_1, h\}, \{a, b\}, \{a, b, \#\}, \delta, q_0, h \}$$

Where δ is defined as follows:

$$\delta(q_0, a) = \{(q_0, a, R), (q_1, a, R)\}$$

$$\delta(q_0, b) = \text{empty}$$

$$\delta(q_1, a) = \text{empty}$$

$$\delta(q_1, b) = \{(q_1, b, R), (h, b, N)\}$$

If the machine has no next move, then it halts without accepting the string.

Remarks 2.2.5.7:

Though we have already mentioned earlier on a number occasions, yet, in view of the significance of non-determinism in designing TMs comparatively *more easily*, we again bring to notice that in the state q_0 on scanning symbol a , the TM may move in any one of the two next possible states viz to q_0 after moving the head to the right or to q_1 (after moving the head to the right). And, if the TM is implemented as a parallel computer then the computer can pursue independently both branches initiated by (q_0, a, R) and (q_1, a, R)

Next, we consider another important variation: Final state Turing Machine Instead of the halt state, TM may have a set F of states designated as final states.

2.2.6 Final State Version of the Standard TM

On the lines of the definitions of finite Automata and Pushdown Automata, we can define (standard) TM also in terms of F , a set of final states, instead of h , the halt state. The only major differences between the TM with F and the TM with h are:

- (i) The TM, while being in a final state, can still have further moves. But in Halt-

state version the TM can not move after reaching the Halt state. In the case of Final state version a TM stops further operations only when there is no next move at a time when the machine is scanning a symbol in some state. If there is no move and the state of TM is a *final state*, then the string on the tape is accepted. However, if there is no move and the state of TM is *not in F*, then TM halts without accepting the string on the Tape.

- (ii) If when the TM is in a final state then the string formed by the contents of the whole tape (excluding the continuous infinite sequences(s) of #'s), is acceptable, irrespective of the position of the Head on the tape. The situation is similar to what we have in case of Halt state version of TM

It can be shown that Final State version of TM is (computationally) equivalent to Halt State Version of TM

With these comments, we give below a formal definition of the Final State version of TM

Definition: Turing machine (Final State Version)

A Turing Machine is a sextuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$

where the various involved symbols denote various entities as follows:

Q	:	The set of states
Σ	:	The set of input symbols
Γ	:	The set of Tape symbols
q_0	:	The initial state
F	:	The set of final states and
δ	:	is a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R, N\}$, with L, R and N respectively denoting <i>move to the Left</i> , <i>move to the Right</i> and <i>No move of the Head</i>

The standard TM and all the extensions of standard TM mentioned above can also be defined in terms of Final State version of the Standard TM on the lines of the above definition.

Ex.2) Construct an NDTM to accept the language
 $\{a^n b^m : n \geq 1, m \geq 0\}$

2.3 UNIVERSAL TURING MACHINE (UTM)

We know the general-purpose computer has the property that the same computer system is used to solve all sorts of problems from different domains of human experience, provided, of course, the problem under consideration is (algorithmically) solvable.

However, from the discussion of Turing machines so far, it is observed that we have constructed a new Turing Machine for each new problem to be solved. On closer examination of the general-purpose computer, we find that the *capability of the computer* in respect of solving any problem, is mainly based on the fact that the program i.e., the description of the sequence of steps (to be executed by the executing component of the computer) in some coded form alongwith the required data, can be stored in the memory of the computer. Later, the control unit of the computer reads the codes for the steps, one step at a time in some order, decodes the code which is read and the concerned executing unit is activated to execute the

corresponding step. This process of *reading* of the code for a step, *decoding* the code and *executing* is repeated till the code for final result is delivered to the memory of the computer.

By following some similar method, even we can construct a (*single*) Turing Machine, which can solve all sorts of solvable problems. Such a Turing Machine is called a Universal Turing Machine (UTM). In order to construct a UTM, let us make the following observations:

Observation I: A Turing Machine M designed to solve a particular problem P , consists, apart from the description of the set of possible states and the set of possible inputs etc. of mainly the description of the process in some coded form of a sequence of steps required to solve the problem in the form of the move-function δ . Thus to solve the problem P , using Universal Turing Machine, the process part involving δ of the Turing Machine M , and the inputs, are expressed in the code (i.e. language) of the Universal Turing Machine. This code of the process (for solving the problem) along with the code of the input, is stored in the memory (i.e., the Tape) of the UTM. And just on the lines of the control unit of a general-purpose computer, the control unit of UTM, reads the codes for steps, one step at a time, decodes and executes the code for each step, until the code for the final result is stored on the Tape of the UTM.

Observation (II): A Turing Machine M designed to solve a particular problem P , can essentially be specified by.

- (i) The initial state say q_{0M} of the Turing Machine M
- (ii) The next-move function δ_m of M , which can be described by the rules of the form: *if the current state of TM M is q_i and contents of cell being scanned are a_j then the next state of M is q_k , the symbol to be written in the current cell is a_l and move m_f of the Tape Head may be :To-Left, To-Right or None.*

Thus, each of these rules for a particular TM M can be specified by quintuples of the form $(q_i, a_j, q_k, a_l, m_f)$. And hence the next-move function δ_m for machine M is completely specified by the set.

$\{(q_i, a_j, q_k, a_l, m_f) : q_i, q_j \in Q_M; a_j, a_l \in \Gamma_M; m_f \in \{ \text{To-Left, To-Right, None} \} \}$

Process part of the TM which is defined by the set of all moves is given by the above set.

Observation 3: Next the question that arises in context of the construction of Universal Turing Machine, is about the number of distinct states in UTM and number of distinct inputs/Tape symbols required in the UTM, so that it can solve any solvable problem.

As UTM should be able to simulate each Turing Machine, therefore, it may appear that number of distinct states and number of distinct Tape Symbols in the UTM, should be at least as much as is possible in any TM, because UTM may be required to accomplish the task of (i.e. to simulate) any TM. However, by proper coding techniques we may use only two symbols to represent set of symbols. This will be shown to be true in a short while. Of Course, if there are enough symbols say for states, then the same symbols may be used for different Turing Machines, if required, just by renaming the states for different TMs.

Though, for each TM, the number of states and the number of Tape Symbols, each is finite for each TM, yet there is no upper bound on each of these numbers.

Therefore, we assume each of the set of states,

$Q_\infty = \{q_0, q_1, q_2, \dots\}$

and the set of Tape Symbols is

$\Gamma_\infty = \{a_1, a_2, a_3, \dots\}$

is countably infinite

The Head-Move set M of the moves of head of course, has only three elements viz, i.e.,

$$H_{mv} = \{L, R, N\}$$

Where L denotes 'Move-Left', R denotes 'Move-Right' and ' N ' denotes 'No Move of the Head'

Observation (IV):

Each of the sets Q_{∞} and Γ_{∞} involves infinitely many symbols. However we cannot produce infinitely many *distinct* symbols required for in the above mentioned entities, viz Q_{∞} and Γ_{∞} . But, we can devise a mechanism to represent these infinite number of distinct entities.

For this purpose, the alphabet set of $\{0,1\}$ of two elements is used to represent all these entities, where sequences of repeated 0's denote various elements of Q_{∞} , Γ_{∞} and H_{mv} . The symbol 1 is used as a separator. Sequences of 1's of different lengths, are used to separate different coded elements.

We will explain these ideas with suitable examples. First, we consider a coding scheme λ for Q_{∞} , Γ_{∞} and H_{mv} in terms of the alphabet $\{0,1\}$, as follows:

$$\lambda(q_i) = 0^{i+1} \quad i = 0, 1, 2, \dots$$

(for example $\lambda(q_0) = 0$, $\lambda(q_1) = 0000$, to be denoted by 0^4 etc)

$$\lambda(a_j) = 0^j \quad \text{for } j = 1, 2, 3, \dots$$

(for example $\lambda(a_2) = 00$, to be denoted by 0^2 ; $\lambda(a_1) = 0000$, to be denoted by 0^4)

Also, $\lambda(L) = 0$, $\lambda(R) = 00$, (or 0^2) and $\lambda(N) = 000$ (or 0^3)

Note that the same sequence of 0's may represent a state, an input symbol or a move, e.g. 000 may represent the state q_2 , the input symbol a_3 and N of moves. However, there is no possibility of confusion or error, because, the strings of 0's are placed in relatively different positions in the representation of a move to denote a state, an input symbol or a move.

Once the basic sets involved in descriptions of the processes, are encoded, we describe the function δ .

We are going to construct UTM as a Deterministic Turing Machine and hence for the move $(q_i, a_j, q_k, a_l, m_r)$ the components q_k , a_l and m_r are uniquely determined by the pair of q_i and a_j and hence we use the shorthand M_{ij} for the move $(q_i, a_j, q_k, a_l, m_r)$.

By the above-mentioned coding scheme, the five components q_i , a_j , q_k , a_l and m_r are respectively represented as $\lambda(q_i)$, $\lambda(a_j)$, $\lambda(q_k)$, $\lambda(a_l)$ and $\lambda(m_r)$, each of which is a sequence of 0's.

Next the move M_{ij} given by $(q_i, a_j, q_k, a_l, m_r)$ may be coded in terms of $\{0,1\}$ by replacing each ',' by one 1 and each parentheses also by one 1.

Thus each move M_{ij} is coded as

$$1 0^{i+1} 1 0^j 1 0^{k+1} 1 0^l 1 0^{\xi} 1,$$

where

$\xi = 1$, if move is to the Left,

$\xi = 2$, if move is to the Right, and

$\xi = 3$, if there is to the 'No Move'.

As, each of the moves will begin and end with a '1', hence, there will be two 1's between two moves. In the representation, therefore, moves are distinguished from its components like states etc

But there is only one 1 between various components of a move. Further, by beginning and ending of the code of a TM marked by three 1's, we distinguish a

TM from its components, i.e, its moves. Also, as mentioned earlier, a Turing Machine is completely specified by the initial state say q_0 and λ the Next-Move function.

In view of these notational conventions, the code of a TM, may be given by

$$111 \lambda (q_0) 1 \lambda (M_{11}) 1 \lambda (M_{12}) 1 \lambda (M_{14}) \dots 1 \lambda (M_{21}) 1 \lambda (M_{22}) \dots \dots 1 \lambda (M_{mn}) 11$$

..... (A)

We may notice that the code of a TM has only two 1's explicitly given at the end of the code. The third 1 is contributed by the code of $\lambda (M_{mn})$, the last move of the machine M. We recall that

$$\Gamma_{\infty} = \{a_1, a_2, \dots\}$$

denotes the set of countably infinite tape symbols and each of the tape symbols a_j , will be coded as

$$\lambda (a_j) = 0^j \quad \text{for } j = 1, 2, 3 \quad \dots \dots \dots (B)$$

The encoding of various code symbols in the (initial) input are separated by 1's, eg, if $a_2 a_4 a_7$ is the initial input then it may be represented as $10^2 10^4 10^7 1$.

Remark 2.3.1:

From the above discussion, we make the following observations, which will play an important role, when later on, we would be giving examples of a language having or not having some properties:

- (i) Every TM can be thought of as a **unique** sequence of binary digits, but only special types of binary sequences, e.g., sequences starting with three 1's.
- (ii) Not a separate observation, but a consequence of observation (i) above but stated separately in view of its significance, is that not every binary sequence represents a TM. Thus every binary sequences can be interpreted as at most one TM
- (iii) In view of (i) and (ii) above, if a binary word w represents a TM M then w treated only as a binary string (and not treated as representation of TM) can also be given as input to the TM M and hence the question 'Does M accept w ?' or 'Does a TM having w as its representation accept w as an input string?' is a relevant question. This question may have a 'yes' answer for some pairs of (M,w) and 'No' answer for some other pairs of (M,w) .

Next, we briefly describe how the UTM will solve a problem P for which a TM M already exists. As a first step, the **process component** of M is encoded in terms of the alphabet set $\{0, 1\}$ as given by (A) above and the (initial) input is encoded using the coding given by (B).

We assume the UTM is a **3-Tape Machine**. The encoding of the *input* for the problem P is written *on the first Tape* of UTM. On the *second Tape* of UTM is written the *process component* of M as is given by (A) above. *On the third Tape*, the *current state of M* is stored. The control unit of UTM simulates the TM M . The control unit by counting number of 0's between 1's, finds out the input symbol a_i on Tape 1 and finds the current state q_i from Tape 3 of UTM. At This stage, control of UTM knows the pair (q_i, a_i) , which uniquely determines the move $M_{ij} = (q_i, a_j, a_k, a_l, m_r)$. The control unit extracts the quintuple $(q_i, a_j, q_k, a_l, m_r)$. From the quintuple, the control unit of UTM extracts q_k , the next state of M ; a_l , the next symbol to be written in the current cell being scanned; and m_r the move of the Head. The control unit of UTM then writes q_k in place of q_i on Tape 3; writes a_l in place of a_j on Tape 1 and moves the Head on Tape 1 of UTM according to m_r . Thus 3-Tape UTM is able to solve the problem P by simulating the solution imbedded in TM M .

2.4 LANGUAGES ACCEPTED/DECIDED BY TM

Problem, its instance and its language:

Let us understand the difference between a *problem* and an *instance of a problem* (sometimes called a *question*) from the following statement:

A **problem** may be to find out the roots of a (*general*) quadratic equation say $ax^2 + bx + c = 0$, with $a \neq 0$, where $a, b, c \in \mathbb{R}$, are *parameters* of the problem. A set of *values* one for each of the three parameters, gives an **instance of the problem** (i.e., a *question*). Thus finding out the roots of a quadratic equation $4x^2 + 3x + 2 = 0$ is an *instance of the problem* of finding the roots of the quadratic equation $ax^2 + bx + c = 0$. Hence, the *problem* of finding the roots of the equation $ax^2 + bx + c = 0$ can be equivalently represented by the set of all triples of the form $(a \neq 0, b, c)$, where each triple, which is just a single string, say $(4, 2, 0)$, represents an *instance of the problem*. Therefore, the problem of finding roots of a quadratic equation $ax^2 + bx + c$ with $a \neq 0, b, c \in \mathbb{R}$ is equivalently represented by the infinite set $\{(a, b, c), a, b, c \in \mathbb{R} \text{ and } a \neq 0\}$, where each member string (a, b, c) , like $(4, 2, 0)$, represents an instance of the problem.

In general a problem is a set of its instances, where each instance is obtained by assigning values to the parameters, from the domain, say D , over which the problem is defined. Thus a problem is equivalently defined as a set from a domain D . Also, each of the element of a domain D can be written as a string over some alphabet. For example, in the case of the problem of finding roots of a quadratic equation, the domain consists of triples (a, b, c) where a, b, c are integers and $a \neq 0$. But each integer can be written as a sequence of digits from the alphabet $\{0, 1, 2, \dots, 9\}$. And hence each triplet can be written as a sequence over the alphabet $\{0, 1, \dots, 9, (,)\}$. Thus, we conclude that each *problem can be thought of as a set of strings over some alphabet*. Also, a set of strings over an alphabet is also called a *language* over the alphabet.

Thus, we further conclude that a *problem can be thought of as a language over some alphabet*.

In the following discussion, unless mentioned otherwise, a language L representing an arbitrary problem P shall be over an alphabet, which we denote by Σ . In other words, a language L will be assumed to be a subset of Σ^* .

For a problem, number of instances *need not always be infinite*. For example, in the problem, of finding roots of a quadratic equation $ax^2 + bx + c = 0$ in which each of $a \neq 0$, and c is a *natural number* less than or equal to 10, then the set of instances or the set of strings representing the problem is 1210, which is finite. However, in context of problems, we are interested, problems generally have *infinite* number of instances, i.e., the sets representing the problems have infinite strings.

Definition: Turing Acceptable Language: A language $L \subseteq \Sigma^*$ is said to be Turing Acceptable language if there is a Turing Machine M which when given an input $w \in \Sigma^*$, such that w also belongs to L , then halts with an output \bar{Y} . However, if $w \notin L$, then M may not halt further if the Turing Machine halts, on an input w with $w \notin L$ then it should halt with an output different from \bar{Y} .

Some authors call Turing Acceptable Language as **Recursively Enumerable language** also.

Definition: Turing Decidable Language: A language $L \subseteq \Sigma^*$ representing a problem over Σ , is said to be Turing Decidable, if there is a Turing Machine M which

always halts when given any input $w \in \Sigma^*$ whether $w \in L$ or $w \notin L$. Further if $w \in L$ then M halts with output \bar{Y} , indicating that the string w is in the language L . And if $w \notin L$, then M halts with output \bar{N} , indicating that w does not belong to L .

Decidable/Solvable Problem: A problem P is said to be Decidable or Solvable if the language $L \subseteq \Sigma^*$ representing the problem is Turing Decidable.
(Some authors call a Turing Decidable language as Recursive set or a Recursive Language.)

Also, we know that an Algorithm is a program that terminates on all inputs. And, also it is not difficult to see that each TM that halts for all inputs can equivalently be expressed as a programme and vice-versa.

Thus, the three statements:

- the statement that a language L is Turing Decidable
 - the statement that language L is a recursive set and
 - the statement that there is an algorithm for recognizing L
- are equivalent.

Note: The phrase recognizing A TM a language is different and more powerful than the phrase "A TM accepting a language"

Remarks 2.4.1: It may be clearly understood that in the case of a language L which is Turing Acceptable Language but which is not Turing Decidable, there may be a TM M which halts on large number of input strings w , where $w \notin L$, but there must be *at least one* string $w \notin L$ on which M does not halt.

Similarly, in the case of a language L which is not Turing Acceptable (and hence can not be Turing Decidable), it may happen that there is a TM M which may halt for a large number of inputs w which belong to L . But there must be *at least one* string $w \in L$ for which M does not halt.

Remark 2.4.2: In respect of the languages defined above, we make the following observations:

- (i) Each Turing Decidable language L is necessarily Turing Acceptable.
- (ii) However, there may be languages which are Turing Acceptable but not Turing Decidable.
- (iii) Further, there may be languages $L \subseteq \Sigma^*$ which may neither be Turing Acceptable and hence nor Turing Decidable. For a language L which is not Turing Acceptable, there can not be any Turing Machine M which halts for every string w of L .

Before discussing properties of the classes of Turing Acceptable languages and Turing Decidable languages, let us mention that we need to consider *at least one example of each of the languages, which is*

- (i) Turing Decidable.
- (ii) Turing Acceptable but Turing Decidable.
- (iii) not Turing Acceptable (and hence not Turing Decidable).

However, the last two required examples form the background of subject-matter of the next section.

Next, we discuss some basic properties of the class of Turing Decidable languages and class of Turing Acceptable languages.

As languages are sets (of strings), therefore, we can talk of union, intersection, and complementation etc. of languages.

Theorem 2.4.3

For two recursive languages L_1 and L_2 , each of the following languages

- (i) $L_1 \cup L_2$
- (ii) $L_1 \cap L_2$
- (iii) $\Sigma^* - L_1$

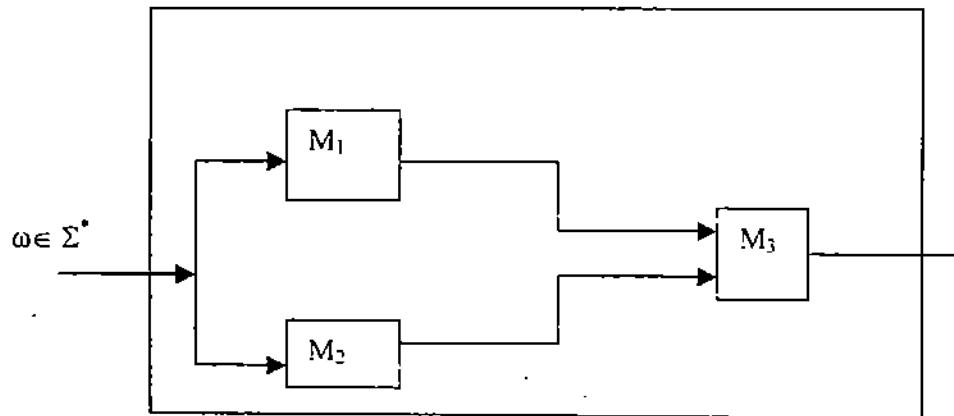
is recursive.

We establish each part of the above Theorem by constructing an appropriate TM deciding the language.

Proof: Let M_i be a TM for deciding the language L_i , for $i = 1, 2$, such that if $\omega \in L_i$, then M_i returns \overline{Y} else returns \overline{N} .

For establishing Part (i) above: we first of all, construct a new Turing Machine M_3 having $\{\overline{Y}, \overline{N}\}$ as the set of symbols. These input symbols are the only possible outputs of each of M_1 and M_2 , and whenever these outputs are available, are written on the Tape of M_3 as inputs to M_3 . The machine M_3 returns \overline{Y} as output, if at least one of the outputs of M_1 or of M_2 is a \overline{Y} .

However, if there is no \overline{Y} in the input to M_3 then the machine returns \overline{N} . The required TM **M-Union** has M_1, M_2 and M_3 as component machines arranged as given by the following figure has The overall control is with the machine M-union.

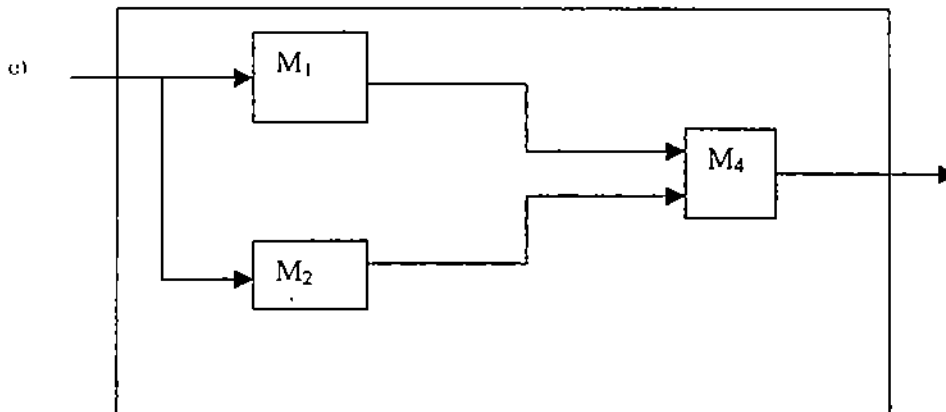


Next, we briefly explain the functioning of the designed machine M-union A string $w \in \Sigma^*$, when given as input to M-union, is further given by the control of M-union, as inputs to both M_1 and M_2 .

As both languages are decidable, therefore, after some finite amount of time, both halt, each with an output as \overline{Y} or \overline{N} . These outputs, whenever delivered are written on the Tape of M_3 . When both the outputs are written on the Tape of M_3 , M_3 is activated. According to the definition of M_3 , it halts with the desired output \overline{Y} if $\omega \in L_1$, or $\omega \in L_2$, else the machine halts with output \overline{N} . The output of M_3 is the output of M-union.

Thus, for each $w \in \Sigma^*$, M-union returns a \overline{Y} or \overline{N} and hence its language $L_1 \cup L_2$ is Turing Decidable.

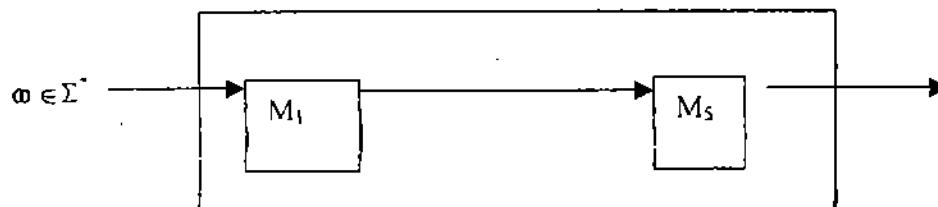
Part (ii) In this case, first of all, we construct a TM M_4 having $\{\bar{Y}, \bar{N}\}$ as set of input symbols. These input symbols, as mentioned earlier, are the only possible outputs of each of M_1 and M_2 . These outputs whenever available are written on the Tape of M_4 as inputs to M_4 . The machine M_4 is designed such that it returns a \bar{Y} if the input sequence consists of both \bar{Y} 's. However, if the input sequence consists of at least one \bar{N} then M_4 returns \bar{N} .



The required TM M-intersection has M_1 , M_2 , and M_4 as component machines as given by the above figure. The overall control also is under M-intersection. The machine functions on the similar lines as M-union functions. The only difference is that its component machine M_4 return \bar{Y} if both M_1 and M_2 return a \bar{Y} , else M_4 returns \bar{N} . And the output of m_4 is the output of M-unit. Thus for each $\omega \in \Sigma^*$, returns either a \bar{Y} or \bar{N} in such manner that if $\omega \in L_1 \cap L_2$ then M-intersection returns a \bar{Y} as output, else \bar{N} as output. Hence its language $L_1 \cap L_2$ is Turing Decidable.

Part (iii): In this case, we construct a TM M_5 which on reading a \bar{Y} returns \bar{N} and on reading an \bar{N} returns a \bar{Y} .

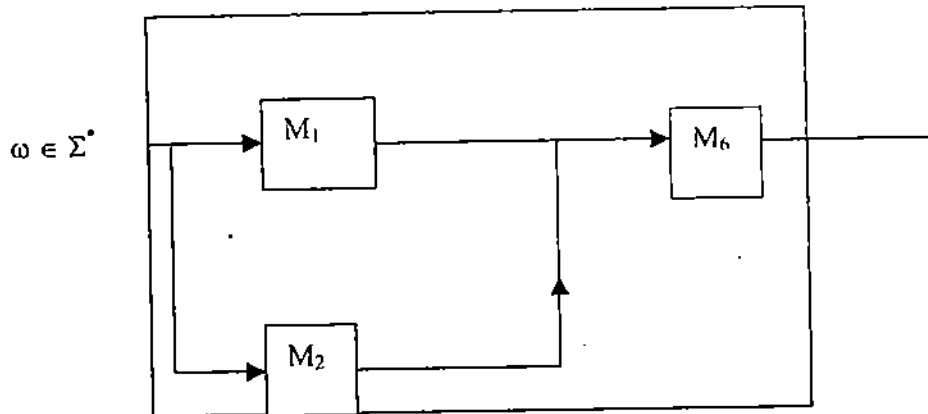
The required TM machine M-complement the following diagrammatic representation.



The machine M-complement functions as follows : When a string $\omega \in \Sigma^*$ is given an input to M-complement, its control passes the string to M_1 as input to M_1 . As M_1 decides the language L_1 , therefore, for $\omega \in L_1$ after a finite number of moves, M_1 outputs \bar{Y} which is then given as input to M_5 , which in turn returns \bar{N} . Similarly, for $\omega \in L_1$, M_5 returns \bar{Y} . Also the output of M_5 is delivered as output of M-complement. Thus for each $\omega \in \Sigma^*$ M-complement returns either a \bar{Y} or \bar{N} s.t. if $\omega \in L_1$ then M-complement returns \bar{N} , else returns \bar{Y} . Hence the language of M-complement is Turing-Decidable.

Theorem 2.4.4: If L_1 and L_2 are recursively enumerable (i.e, Turing Acceptable) languages then $L_1 \cup L_2$ is also recursively enumerable.

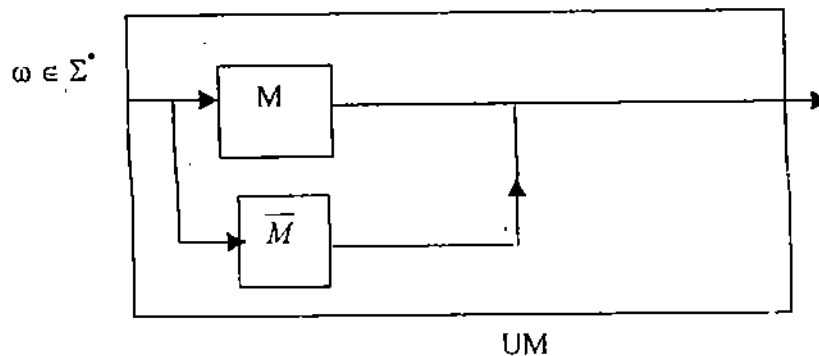
Proof: Let $M_i, i = 1, 2$, be TM, that accepts all strings $\omega \in L_i$, but may or may not halt if $\omega \notin L_i$.
 Then a TM M -A-union with the following configuration and description accepts $L_1 \cup L_2$.



The overall control in M-A-Union which may stop and start any or all of M_1, M_2 and M_6 . The TM M_6 functions as follows: If, at any stage, there is an output from any one of M_1 or M_2 , then on the first output from either M_1 or M_2 , the machine M_6 is activated and the output from M_1 or M_2 , whichever is available, is written on the tape of M_6 . If the output is \bar{Y} either from M_1 or M_2 , say M_1 , then the control of the overall machine M returns a \bar{Y} and halts the machine. However, if it is an \bar{N} , say from M_1 , then the other machine M_2 and hence the overall machine M continue operations. If at any later stage, the other machine, which we have assumed is M_2 , halts and M_2 halts with a \bar{Y} , then overall machine M gives the output \bar{Y} and Halts. If M_2 halts with an \bar{N} , then \bar{N} is returned. However, if either none of the two machines M_1 or M_2 halts, or one of the machines halts with output \bar{N} but the other machine does not halt then, the overall machine continues its operations without halting.

Theorem: For a given language L , if both the languages L and $\bar{L} = \Sigma \sim L$ are Turing Acceptable, then L is Turing Decidable.

Proof: Let M and \bar{M} be the TMs that accept respectively the languages L and \bar{L} . The overall machine UM with following configuration and description will, as we will show, be able to recognize/decide the language L , thereby establishing that L is Turing Decidable.



Whenever an input string $\omega \in \Sigma^*$ is received by UM , its control unit writes ω on the tape of both the machines M and \bar{M} and activates both M and \bar{M} . Whenever an output if at all, comes out of M or \bar{M} then the overall machine UM gives output and Halts.

After following actions: If $w \in L$ then M halts with output \bar{Y} . In this case the overall control returns \bar{Y} as the output of UM. Further, if $w \notin L$ then M halts and returns \bar{N} . The overall control on checking a \bar{Y} from \bar{M} returns \bar{N} as output of UM. Thus for $\omega \in \Sigma^*$, the machine UM always halts and returns \bar{Y} if $\omega \in L$ and returns \bar{N} if $\omega \notin L$. Thus UM decides the language L . Therefore, L is a Decidable language.

When a problem is said to be (formally) solvable/unsolvable?

The issue of solvability/ unsolvability of some of the problems like *squaring a circle* have been occupying the attention of the scholars since time immemorable. In the recent times, *Fermat's Last Theorem* and *Four Colour Problem*, though solved, have been occupying attention of the researchers/scholars in the concerned discipline. Also, now computers are being used as tools for helping the human beings in attempting solutions of problems. Thus, it is very important to know **what in formal sense we mean by a solution of a problem**. We discuss the issue briefly here. However, the issue is the main topic of discussion in a later unit.

From our earlier discussion, we know that each problem may be represented by a language say L . Then we say a problem P is an **unsolvable problem** if the language representing the problem is *not decidable*, i.e., no Turing Machine can be designed which decides the language L corresponding to the problem: P . The following problem, which is quite simple in description, is one of the problems, which is a well-known unsolvable problem.

Unsolvable Problem:

The Halting Problem: Given an arbitrary machine M and a string ω , does M halt with ω as input string?

Remark 2.4.5:

The above problem is mentioned just to show how the concepts of Turing Decidable and Turing Acceptable machines are related to problem solving. However, the proof of the above claim about unsolvability of Halting Problem and general discussion of solvable/unsolvable problems will be subject matter of a later unit.

Remark 2.4.6:

Though the proof of the above claim will be taken up in a later unit, however, briefly, we will like to tell here what is meant by an unsolvable problem through the example of Halting problem. In this context, it may be stated that there can be large number of TMs, in case of each of which it is possible to tell whether it will halt on particular strings or not. *But, if the Halting Problem is unsolvable, then given a general TM and an arbitrary input string, it is not possible to tell whether the TM will halt or not.* The situation is somewhat similar to saying that there is *no systematic method* which can solve an equation of degree 5 or more. But for equation of the form $x^5 - a = 0$ or $ax^{10} + bx^5 + c = 0$, there are systematic methods which can solve equations of degree greater than or equal to 5. **But still we say the problem of finding roots of an equation of degree 5 or more is unsolvable.**

Ex. 3) Show that the language $L = \{ a^n b^n c^n \mid n \geq 0 \}$ is Turing Decidable, showing thereby that every decidable language need not be a context-free language.

2.5 THE DIAGONAL LANGUAGE AND THE UNIVERSAL LANGUAGE

2.5.1 : Definitions of the Languages

In continuation of our discussion, in Section 2.3, about representation of TMs as binary strings, we discuss two very important, but not intuitive, languages which provide important examples for languages having some particular properties but not having some other properties. The languages are

- (i) L_d , the language of strings w , where each string w in L_d is such that w is not acceptable by TM M having the string w as its representation. L_d also includes those strings w which are not binary representation of any TM. For example, as representation of every TM, by our construction in Section 2.3, must have '111' as leading part of its representation as a binary string, therefore the binary string '00' is not a representation of any TM and hence the string 00, not being representation of any TM, can not accept any binary string w and hence 00 also belongs to L_d

In literature, L_d is also known as NSA *not self-accepting* and by some other names. The suffix *d* stands for *diagonalization*, the significance of which will be explained later.

- (ii) L_u , the set of all binary string α , where α represents the ordered pair (M, w) where M is a Turing Machine and w is any binary string such that M accepts w . In other words $\alpha = (\alpha_1, \alpha_2)$ is some suitable binary representation of $\langle M, w \rangle$, where α_1 is a binary representation of a TM M and $\alpha_2 = w$ is a binary string and M accepts w . L_u is also the language representation of what is known as **Halting Problem for Turing Machine**.

Explicitly, *Halting Problem states*: Is it possible to tell, for an arbitrary TM M and an (arbitrary) input string w , whether M accepts w ?

The answer to the **Halting problem** is **no**, and we discuss the problem in detail later. The suffix *u* in L_u stands for *universal*.

The following two important questions arise about the two languages viz L_d and L_u defined above

- (i) Can we show the existence of each of L_d and L_u by some constructive methods?
(ii) Is each of the two languages L_d and L_u Turing Decidable? And if any of these is not Turing Decidable, then is that language Turing Acceptable?

First of all, we answer the **Question (ii) above without justification**. Justification for our answer will be given after a while.

- The language L_d is not Turing Acceptable (and hence not Turing Decidable).
- The language L_u is Turing Acceptable but not Turing Decidable.

2.5.2 Constructive Existence of L_d

From Section 2.3 on Universal Turing Machine, we know that each Turing Machine can be represented by a **finite** string over $\{0, 1\}$. In order to show the existence of L_d and L_u by constructive means, we discuss a method of enumerating all TMs, i.e., listing all TMs by some ordering of their binary representations. For this purpose, we

define a rule which gives a sequence for representations of TMs, in which a particular representation follows an already enumerated representation, if any.

By a similar method, we can enumerate all input strings w over $\{0, 1\}$.

We make a list of binary representations of all TMs constructively as follows:
First, we take all binary strings of length 0, then we take all binary strings of length 1; followed by all strings of length 2 and so on.

For distinct strings say s_{i1} and s_{i2} of length i , we find out the decimal numbers d_1 and d_2 having s_{i1} and s_{i2} as binary representations. Then, in our listing, s_{i1} precedes s_{i2} , iff $d_1 < d_2$. Thus all binary strings representing TMs are listed in an order which is generally called **lexicographic order**.

The ordering of TMs is as follows:

- (i) Take one by one binary strings in the lexicographic ordering defined above.
- (ii) For the chosen string α , check whether it represents a TM according to coding defined in Section 2.3. If α does not represent a TM, take next string from the list and go to Step (ii). If α represents a TM, follow the next step.
- (iii) If α represents a TM, then α is put at the end of the list containing members of the list already obtained by the process. And then take next string from the listing of strings and go to Step (ii) above.

This is called enumeration of TMs. After the above discussion, all TMs can be listed as T_1, T_2, T_3, \dots according to lexicographic listing of their binary representations. Similarly, all input strings can also be listed as w_1, w_2, w_3, \dots . We have already explained that, in general, how any finite or infinite set of binary strings, where a string may or may not be representing a TM or may or may not be representing an input w , can be lexicographically ordered.

Constructive Existence of L_d , the language of all those strings w s.t

- w represents a Turing Machine say T_i , and further,
- If w is given as input to T_i , then T_i does not accept w .

The construction of L_d is three-step process:

Step (i): Make a Table of the form with row-headings as T_i in the order defined above and column headings as w_j the binary strings, which are also lexicographically ordered, and which may be given as inputs to T_i .

At this stage, the table may appear as

	w_1	w_2	w_3	...
T_1				
T_2				
T_3				

(In the above table T_i may be a hypothetical TM, which actually do not represent any TM. In such cases, for any string w , we say T_i does not accept w , where w may be any string. The complete row for such a T_i consists of 0's only.)

Step (ii): Next we fill up entries of the table as follows. The entry (T_i, w_j) is 1 if T_i accepts the string w_j and the entry (T_i, w_j) is 0 if T_i does not accept w_j .

Thus, let us assume we get a table of the form

	w_1	w_2	w_3	w_4
T_1	1	0	0	1
T_2	0	1	0	1
T_3	1	1	0	0
T_4	1	0	1	1

Step (iii): Next we construct the language L_d as

$$L_d = \{u_1, u_2, \dots, u_k, \dots\}$$

where string u_k is obtained from the row labeled T_k in the above table, by inverting its k th bit and keeping all other bits unchanged. For example, $u_1 = 0001$..

Which is obtained from the row labeled as T_1 by inverting the bit in (1, 1)th position. Similarly

$u_4 = 1010$, which is obtained by changing (4, 4)th entry of the row labeled with T_4 as row-heading.

This completes the construction of L_d

Remark 2.5.2.1:

The process of obtaining L_d is by replacing the values of the diagonal elements by any value different from the earlier value. This is why, the process is also called diagonalization.

Diagonalization is an important method of showing that a language does not have a particular property. The method was devised by the well-known mathematician Georg Cantor (1845-1918) and used the method in 1895 to show that not every real number is a rational number.

2.5.3 Constructive Existence of L_u

L_u is the language of strings of the form α , where α represents an ordered pair (α_1, α_2) with α_1 , a binary string, representing a Turing Machine say M_1 and α_2 , some binary word, such that M_1 accepts α_2 .

Once α_1 and α_2 are known, by an appropriate binary encoding scheme for making ordered pairs out of binary strings, it can be easily seen that $\alpha \in L_u$ is a binary string. Further the strings within L_u are enumerated by Lexicographic ordering. This completes the listing process for the elements of L_u .

2.5.4: The Diagonal Language is not Turing Acceptable

Remarks 2.5.4.1:

Before we go ahead with the proof of properties of L_d and L_u , it is interesting, and will be later on useful also, to consider sets representing similarity and differences between elements of L_d and L_u , the complement of L_u

$$L_d = \{w : w \text{ is not acceptable by the TM having } w \text{ as its binary code}\}$$

$$= \{w : w \text{ is not a representation of any TM}\} \cup \{w : w \text{ is binary code of a TM say } M_j \text{ but } M_j \text{ does not accept } w\}$$

$$L_u = \{ \langle M, w \rangle : \langle M, w \rangle \text{ is binary code representing the pair } (M, w) \text{ where } M \text{ is a Turing Machine that accepts } w \}$$

Therefore

$\overline{L_u} = \{ \alpha : \alpha \text{ is a binary string s.t either } \alpha \neq \langle M, w \rangle \text{ or if } \alpha = \langle M, w \rangle \text{ then } M \text{ does not}$

accept } $w \}$

$= \{ \alpha : \alpha \text{ does not represent ordered pair of a TM and an input string} \} \cup \{ \alpha : \alpha = \langle M, w \rangle \text{ and } M \text{ does not accept } w \}$

Remark 2.5.4.2:

We may note there is parallel between each pair of languages L_u and $\overline{L_u}$ and the languages $\overline{L_d}$ and L_d . However, differences between languages within a pair are of the form of inputs:

- (i) A member of L_d is a string w which represents just the input to the Turing Machine M , which, if exists, does not accept w . Therefore, there is inbuilt system which finds out whether such an M exists or not
- (ii) However, $\overline{L_u}$ is though again a binary string α , yet it represents (M, w) , i.e, there are two distinct parts in α , first part of α is expected to represent a TM M and the rest of the part an input string w to M s.t M does not accept w .

The first part of α may not represent a TM and then automatically $\alpha \in \overline{L_u}$ without any further the T.M, which failed to exist.

The main difference between L_u and $\overline{L_u}$ is that a member of L_u represents only inputs w to TMs whereas the each member of L_u is a binary string the form $\langle M, w \rangle$ in which first part is expected to represent a TM and second part an input to TM.

Similar are the difference between $\overline{L_d}$ and L_d

Next, we prove that the statements made earlier about L_d

Theorem 2.5.4.3: The language L_d is not Turing Acceptable (or equivalently L_d is not recursively enumerable)

Proof: The theorem is proved if we are able to show that there does not exist a TM which accepts the language L_d . Now the proof follows from the following facts which we came across during the construction of L_d :

- (i) All possible Turing Machines are listed as row-labels in the table constructed for the definition of L_d . Thus if there is a TM that accepts L_d then it must be label of some row i.e, must be some T_i which a row-label of the table.
- (ii) L_d by its construction, differs from the machine T_k in the k th position, for all k . In other words $L_d \neq T_k$ for all k .

Therefore, there can not be any TM that accepts L_d

Remark 2.5.4.4:

Proving of L_d as not Turing Acceptable, by itself, may not appear to be a great achievement in the sense that L_d is a highly contrived unintuitive language. The significance of L_d not being Turing Acceptable, lies in the fact that, it is used in establishing non-Turing-Decidable/acceptable character of a number of languages, which are not so unintuitive. We will discuss a number of Turing non-decidable or undecidable languages and problems in Block3. At present we discuss properties of the universal language L_u

2.5.5 L_u Turing Acceptable but not Decidable

Theorem 2.5.5.1:

The language L_u of all binary strings a representing those pairs of arbitrary TMs M and arbitrary input strings w for which M accepts w , is Turing Acceptable but not Turing Decidable.

May be used as justification for the Halting problem is undecidable.

Proof: The proof consists of two parts

- (i) L_u is Turing Acceptable
- (ii) L_u is not Turing Decidable

L_u is Turing Acceptable: A language L is acceptable if we are able to design a TM \underline{M} that accepts L . We only sketch below the design of the required TM \underline{M} , which, designed on the pattern of a Universal Turing Machines, is a three-tape TM. For a given TM M and an input string w , the following steps are taken to return a yes, if M accepts w :

Step 1 (a) The binary code of M followed by the input string w is placed on Tape 1 which is only read, but not written, to guide simulation of the behavior of machine M on input w .

(b) The Tape 2 is used for simulating the behaviour of M on w as input.

Initially Tape 2 is written with the string $\# w \#$.

Tape 3 contains the state of M , during the process of simulation of M by \underline{M} . Initially, q_0 , the initial state, is written on Tape 3.

Step 2: The Process of Simulation of M by \underline{M}

At any time, the Head of Tape 2 scans a cell of Tape 2 and hence, knows its contents v at any point of time in the process of simulation of M . The control of \underline{M} also knows the state q of the simulated machine M from Tape 3. From the known pair (q, v) the control of \underline{M} finds from Tape 1 the value (p, u, m) s.t. $\delta_M(q, v) = (p, u, m)$ where δ_M is the next-move function of M . At this stage, the control of \underline{M} takes the following actions:

- (i) replaces the contents of the currently scanned cell of Tape 2 from v to u . And moves the Head of Tape 2 according to the move m ;
- (ii) changes the contents of Tape 3 to represent the new state p by replacing the representation of the previous state q .

If M accepts w , then the whole process is repeated till we reach halt state of M in which case the control of \underline{M} returns 'yes' and if required, waits for the next (M, w) pair to be written on Tape 1 and whole process is repeated.

The language L_u is not Turing Decidable:

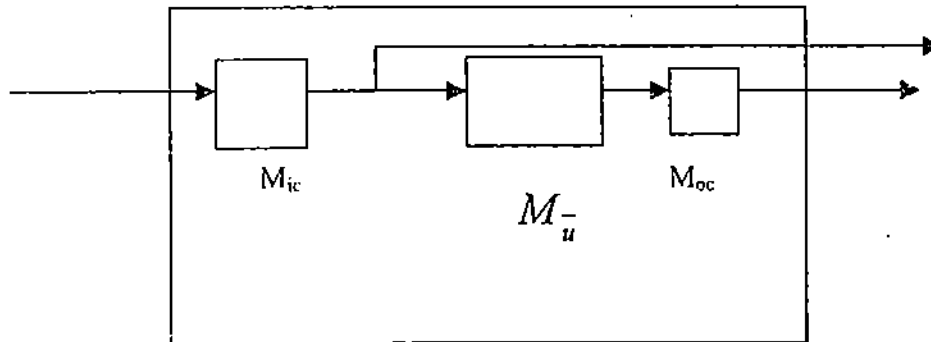
We prove the above-mentioned statement by contradiction. Let L_u be decidable. Then, by definition, $\overline{L_u}$, the complement of L_u , is Turing acceptable. But, then we show below that (Turing) acceptability of $\overline{L_u}$ implies acceptability of L_u . But we know L_u is not acceptable. Hence we arrive at a contradiction, leading to the fact that the assumption is wrong. Therefore L_u would be undecidable.

Next we show $\overline{L_u}$ is acceptable $\Rightarrow L_d$ is acceptable.

If $\overline{L_u}$ is acceptable then there must be a TM say M_u that accepts the language $\overline{L_u}$.

We intend to design a TM M_d which accepts L_d using M_u as a component as shown below.

(But, M_d otherwise should not exist as L_d has already been shown to be not acceptable)



M_d consists of three parts

- (i) M_{ic} which converts L the input to M_d , into an input to M_u . In other words, M_{ic} converts a member of L_d into a member of $\overline{L_u}$
- (ii) M_u , being a Deciding machine for L_u returns a 'Yes' or 'No' on each input w , irrespective of whether $\alpha \in L_u$ or $\alpha \notin L_u$
- (iii) M_{oc} then converts this Yes/No into an appropriate response of M_d to α , as input to M_d

As M_u is assumed to be already designed TM that decides language L_u , therefore, if we are able to explain the designs of M_{ic} and M_{oc} then M_d will be designed.

Also, we have already explained that L_d and $\overline{L_u}$ represent equivalent languages except the form of its members. Therefore, responses of machines M_d and M_u must be same the on corresponding inputs. Therefore, M_{oc} is an identity machine that returns the input as output.

Hence we are left with the design of M_{ic} , which we accomplish as follows:

Let α be an input to M_d , the (hypothetical) machine that accepts the language L_d . Therefore α must be treated as a binary string which is to be given as an input to the TM, which if it exists, has α itself as its code. As all TMs are lexicographically coded, therefore an algorithm can be designed to find out whether α is a code for a TM or not. If α is not the code of a TM, then it can not accept itself as input and hence the question 'Does α reject α ?' has answer yes. Therefore, we may give the output of M_d as accepted or Yes without feeding it to M_u .

If α is the code of some TM say M , where M is found by the step, explained in previous paragraph, then the code for the ordered pair M and α is given as input to M_u . This completes the construction of M_{ic} and hence of M_d which decides L_d if M_u decides $\overline{L_u}$.

But, as L_d is Turing undecidable, there can not be a TM M_d deciding it. Hence no M_d deciding L_d can exist leading to the conclusion that L_d is undecidable.

Remark 2.5.5.1:

The proof given above in support of the truth of the statement ' L_d is not Turing Decidable', may without any change, be given in support of the truth of the statement: **Halting problem is undecidable.**

2.6 CHOMSKY HIERARCHY

In the previous units of the course, we discussed languages, i.e, sets of strings each over a (finite) alphabet from at least two different perspectives:

- (i) Languages accepted by automata viz accepted by Finite Automata, by Pushdown Automata and by Turing Machines.
- (ii) Languages generated by formal grammars viz by a context-free

Informally, a **grammar** is a notation for specifying/defining its language through a finite number of rules.

To have an idea of what a grammar is in the *formal* sense, we recall the definition of a context-free grammar. (In the literature, there are many variations of the following definition).

A context-free grammar of a language is given by

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

where V is the set of variables, T is the set of terminals, S the start symbol and P is the set of productions of the form

$$A \rightarrow \alpha$$

and where $A \in V$, the set of variables and $\alpha \in (V \cup T)^*$, i.e., α is a string, possibly empty, of variables and terminals.

In the formal sense, a general grammar G may be defined as a four-tuple

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

The three components viz V, T, S may be the same for various types of grammars. However, it is the form of productions that distinguishes the types of languages. Chomsky is among the first in the modern times to have introduced the concepts of formal grammar/language. However, the idea of defining languages through formal grammars was used many centuries before Christ, by Panini, a Sanskrit scholar, in defining Sanskrit language through formal grammars.

Chomsky through his papers, defines four classes of languages and named these classes as Type 0, Type 1, Type 2, and Type 3, such that each language of type $(i + 1)$ is also a language of Type i , but converse does not hold. However, now-a-days, these classes are better known by other names. For example,

The type 0 languages are better known as recursively enumerable languages, or as phrase-structure languages or sometimes as semi-free and even as unrestricted languages.

*Chomsky N : Three Models for the Description of language, *IRE Transactions on Information Theory* 2: 113-124, 1956

*Chomsky N : On certain Formal Properties of Grammars, *Information and control* 2: 137- 167, 1959

The **type 1 languages** are known as **context-sensitive languages (CSL)**

The **type 2 languages** are known as **context-free languages (CFL)**. Finally, the **type 3 languages** are called **regular languages**. Another type of languages, which is not mentioned under the Chomsky Hierarchy is the type of recursive languages of Turing Decidable languages, which as per definition given earlier, are the languages L over alphabet Σ each of which a TM T can be designed which halts for every string $\alpha \in \Sigma^*$, irrespective of whether $\alpha \in L$ or $\alpha \notin L$.

Also, we may notice that out of the five types of languages mentioned above, we have come at some stage or other all the types except the type of context-sensitive languages (CSL).

Also, the Linear Bounded Automata (LBA) which corresponds to CSL is also a new type of automata.

Next, we define a (formal) grammar for each type of languages (under Chomsky hierarchy). Then we mention one-to-one correspondence, between these languages and different types of automata and in the process introduce a new type of automata in order, to make the one-to-one correspondence complete. Also we discuss closure properties of the various types of languages.

Grammars for languages under Chomsky Hierarchy*

Regular languages (Type 3 languages)

So far, we know that a language L is regular if

- i. L is accepted by a Finite Automata or
- ii. L can be expressed by a regular expression

Also, we know that a regular language is a context-free language and a context-free language can be described by a context-free grammar. Thus, a regular language may be definable by some special context-free grammar. Actually, a regular language is characterized by a special context-free language called **regular grammar** to be defined below. However, for this purpose, we need the definitions of **right-linear grammar** and **left-linear grammar**.

Right Linear Grammar: A context-free grammar

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

is said to be right linear if every production in P is of the form

$A \rightarrow a$ or $A \rightarrow aB$,

Where A and B belong to V , the set of variables; and a belongs to T , the set of terminals. S , of course, is the start symbol.

Definition Left-Linear Grammar: A context-free grammar

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

is said to be left linear if every production in P is of the form

$A \rightarrow a$ or $A \rightarrow Ba$

Where A and B belong to V , the set of variables and $a \in T$, the set of terminals. S is the start symbol.

Definition Regular Grammar: A context-free grammar

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

is called regular, if it is either left-linear or it is right-linear.

* refer PP. 327-330 Introduction to Languages and the Theory of computation by John C. Martin (TMH, 1998)

Example 2.6.1.1: The regular language $L = \{a^n : n \geq 1\}$ over $T = \{a\}$ has the regular grammar given by

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow aA \end{aligned}$$

We have already studied the context-free grammars and context-free languages (Types 2 languages) in detail. Also we know the equivalence of pushdown automata to context-free languages. Therefore, we skip to next type of languages.

Next we introduce context-sensitive grammars, context-sensitive languages (type 1 languages) and then we discuss linear bounded Automata, all three of which are new concepts.

Definition: Context-Sensitive Grammar A grammar $G = (V, T, P, S)$,

where V is the set of variables; T is the set of terminals, P is the set of productions and S is the start symbol, is said to be context-sensitive grammar, if every production is of the form

$$\alpha \rightarrow \beta$$

Where

- (i) $\alpha, \beta \in (V \cup T)^*$
- (ii) $|\beta| \geq |\alpha|$, where $|x|$ denotes number of letters in the string x
- (iii) α Contains at least one variable

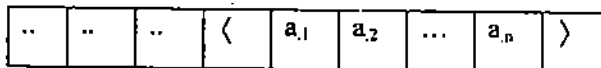
Definition Context-Sensitive Language: A language generated by a context-sensitive grammar is called a context-sensitive language

Example 2.6.1.2:

We just mentioned, without actually producing a grammar, that many of the programming languages including Pascal and C are not context-free, but are context-sensitive languages. These languages are not context-free because of the need for defining of CSL.

Definition: Linear Bounded Automata (LBA) is an NDTM $M = (Q, \Sigma, T, S, q_0, h)$ with the following restrictions:

- (i) Two special symbols viz \rangle and \langle , not belonging to Γ , are written on the tape along with the input string $x = a_1 a_2 \dots a_n$ in the following manner



In other words the symbols \rangle and \langle are respectively used as the initial right- end marker and left-end marker of input string.

- (ii) Tape head may scan the cells containing \rangle and \langle but, these cells can not be written into
- (iii) Tape head cannot move to or scan any cell right of \rangle and any cell to the left of \langle

Remark 2.6.1.3:

In other words, a Linear Bounded Automata is a restricted Non-Deterministic Turing Machine, which does not have potentially infinite tape as working space for (intermediate) computations: Rather working space is restricted to the finite number of cells containing the (initial) input and the cells of the two end-markers.

Remark 2.6.1.4: In view of the statements above that

- (i) Context-sensitive languages can alternatively be defined as the languages accepted by LBAs
- (ii) Every context-sensitive language need not be context-free language, we conclude that LBA is more powerful automata machines than pushdown automata.

Next, but not finally, we consider grammars for recursively enumerable language, (Type 0) i.e, languages accepted by TMs. These grammars are generally known as unrestricted grammars or phrase-structure grammar.

Definition : Phrase-structure/unrestricted Grammar:

A grammar

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

is said to be phrase-structure unrestricted grammar, if P consists of productions of the form

$\alpha \rightarrow B$,

where

- (i) α, B are strings over Σ and
- (ii) α contains a variable.

As usual, the letters V, T and S respectively denote set of variables, set of terminals and the start symbol.

Next we discuss a type of languages, which does not fall under any of the four types of languages covered by Chomsky hierarchy, viz recursive language or recursively decidable language.

We recall that a **recursive language** L is language over some alphabet say Σ , for which there is a TM M such that for each string $x \in \Sigma^*$, M halts and further

- (i) M halts and returns Y (for yes) for each $x \in L$ and
- (ii) M halts and returns N (for no) for each $x \notin L$

However, so far recursive languages have not been characterized by any grammars.

Subject set-Superset Relationship between types of Grammars/Languages

In the earlier units, we have proved or stated that

- (i) Each regular language is a context-free language but the converse need not be true. For Example, the language $\{a^n b^n : n \geq 0\}$ is **context-free but not regular**
- (ii) Each context-free language is recursive (i.e, Turing decidable) but the converse need not be true.

For example the language $L = \{a^n b^n c^n : n \geq 0\}$ is **not context-free but is Turing decidable** (i.e, is recursive) language.

- (iii) From the definitions of context-free grammars (CFG) and context-sensitive grammar CSG, it is clear that every, CFG is also CSG and hence every CFL is CSL also. However, converse is not true. We have already mentioned that the programming languages including Pascal and C are not context-free but context-sensitive languages.
- (iv) It is beyond the scope of the course, but by using diagonalization method indirectly or directly it can be proved* that
 - (a) Every context sensitive language is recursive (or Turing decidable)
 - (b) There is a recursive language, which is not context sensitive language.

Moreover, a recursive language containing null string, can not be a context-sensitive language

- (v) In the previous unit, we mentioned that every recursive/decidable is Turing acceptable but the converse need not be true

Thus if we use the notations

L_R : the set of all recursive languages
 L_{CF} : the set of all context-free languages
 L_{CS} : the set of all context-sensitive languages
 L_{REC} : the set of all recursive languages
 L_{PH} : set of all phrase-structured/Turing acceptable recursively enumerable languages, then we have the following set relationship:

$$L_R \subseteq L_{CF} \subseteq L_{CS} \subseteq L_{REC} \subseteq L_{PH}$$

Closure properties of various types of languages under standard set operations

Definition : By closure property of a set of languages L_P having property $-P$, under an operation say op means that if L_1 and L_2 are two languages in L_P then $L_1 op L_2$ is also in L_P .

Many of the following properties have been derived in the earlier units. The rest of the properties are just mentioned below without any proof. Interested reader may refer to martin (1998) and Hopcroft and Ullman (1979, 1987).

- (i) L_R , the set of all regular languages, is closed under all standard set operations, viz under intersection, union, complementation, concatenation and Kleene star.
- (ii) L_{CF} , set of all context-free languages, is closed under union, concatenation and Kleene star, but is not closed under intersection and complementation.
- (iii) L_{CS} , the set of all context-sensitive languages, is closed under union, intersection, concatenation and complementation.

However, as a language containing null string can not be a context-sensitive language, therefore, for a context-sensitive language L , the language L^* can not be context sensitive but, it has been proved that if L is context-sensitive then L^+ is also context-sensitive where L^+ is the set of all strings obtained by concatenating all finite, but at least one, number of strings from the language L .

2.7 SUMMARY

In this unit, we discuss various extensions of the standard TM that was defined in Unit 1 and state facts of their equivalences to standard TM. Each TM is designed to solve one problem (i.e, one type of questions). However, Universal Turing Machine, which also is defined and explained in this unit, is like a general-purpose computer, and hence is capable of solving *any* problem, provided that the problem is solvable by computational means. Next, we explain how a problem can be thought of as a language and how a language is accepted decided by a TM, and in the process, how a problem is solved by a TM.

2.8 SOLUTIONS/ANSWERS

Exercise 1

Part (1): To convert $\# w \#$ into $\# w \# w \#$

Hint : In stead of the δ -move under (*) of Step 3 of Example 2.2.3.1, in this case, we have

$$\delta(q_3, \#, \#) = (q_4, (\#, R), (\#, R))$$

Rest of the steps are the same as in Example 2.2.3.1

Part (ii) : To covert $\#w\#$ into $\#w^R\#$

Hint : After executing steps 1 and steps 2 of Example 2.2.3.1 in Step 3 we

move the Head of Tape 2 towards left and the Head of Tape 1 towards right as follows

$$\delta(q_2, \#, \#) = (q_3, (\#, N), (\#, L))$$

$$\delta(q_3, \#, \#) = (q_3, (\#, R), (\#, L))$$

(Copying symbols of Tape2 to Tape1 in reverse order)

$$\delta(q_3, \#, \#) = (\text{Halt}, \#, \#)$$

Part (iii): To convert $\#w\#$ into $\#w\#w^R\#$

Hint : In stead of the following δ -move of part (ii) above,

$$\delta(q_2, \#, \#) = (q_3, (\#, N), (\#, L))$$

we have the following move

$$\delta(q_2, \#, \#) = (q_3, (\#, R), (\#, L))$$

Rest of the δ -moves are the same as in step (ii)

Exercise 2: The required TM $M = \{Q, \Sigma, \Gamma, \delta, q_0, h\}$, with

$Q = \{q_0, q_1, h\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, \#\}$ and δ being given by the following Transition Diagram

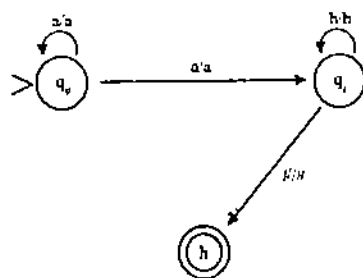


Fig 1.8.1

- (i) the TM halts in q_0 , if the first symbol is not an a
- (ii) the TM halts in state q_1 , if it finds an a after already having scanned h.
- (iii) In state q_0 on scanning a, the TM activates two branches, viz, one in state q_0 and the other in state q_1 . If the next symbol happens to be an a then

the q_1 -state branch dies and only q_0 -state branch remains alive. The rest of the behavior of the TM is apparent from the figure above

Exercise 3: In order to show L as Turing Decidable we need to design a TM that accepts both the language

$$L = \{a^n b^n c^n : n \geq 0\}$$

and the language $\Sigma^* \sim L$, we construct a 3-tape TM M as follows:

$$\text{Let } M = \{Q, \Sigma, \Gamma, \delta, q_0, h\}, \Sigma = \{a, b, c\}$$

$$\Gamma = \{a, b, c, \#\}$$

The sequence of steps for defining δ for the required TM is, as given below

Step 1: Write any given string w over $\Sigma = \{a, b, c\}$ on Tape 1 as $\# w$ and the TM is activated in state q_0 where all heads, e.g., H1, H2 and H3 of respectively Tape 1, Tape 2 and Tape 3 are scanning of the left-most cells the respective tapes.

Step2: Copy the contents of Tape 1 to Tape 2 and Tape 3 through the following moves:

$$\delta(q_0, (\#, \#, \#)) = (q_1, (\#, \#, \#), (R, R, R))$$

$$\delta(q_1, (x, \#, \#)) = (q_1, (x, x, x), (R, R, R)) \quad \text{for } x \in \Sigma$$

$$\delta(q_1, (\#, \#, \#)) = (q_2, (\#, \#, \#), (L, L, L))$$

$$\delta(q_2, (\#, \#, \#)) = (h, (\#, \#, \#), (N, N, N))$$

(null the string case acceptance)

At this stage, all Heads are scanning right-most non-blank cells if any, of respective tapes.

Step 3: From the right most non-blank cells on three tapes, we reach the right-most cell of Tape1 that contains a , if any, and reach right-most cell of Tape2 that contains b , if any, and Tape 3 is not moved, by making the following moves

$$\delta(q_2, (c, c, c)) = (q_3, (c, c, c), (L, L, N))$$

$$\delta(q_3, (b, b, c)) = (q_3, (b, b, c), (L, N, N))$$

$$\delta(q_3, (a, b, c)) = (q_4, (a, b, c), (N, N, N))$$

At this stage H₁ Head should be scanning right- most a on Tape1; Head 2 should be scanning

right-most b on Tape2 and Head 3 should be scanning right-most C on Tape3. Further, for strings in L , we do not expect c to the left of any b on Tape 2 and no b or c to the left of any a on Tape 1.

Step 4: Next we match number of a 's on Tape1, to number of b 's on Tape2 and number of c 's on Tape3 through the following moves:

$$\delta(q_4, (a, b, c)) = (q_4, (a, b, c), (L, L, L)) \quad \text{and}$$

$$\delta(q_4, (\#, a, b)) = (h, (\#, a, b), (N, N, N))$$

If we reach the Halt state h through the above-mentioned moves, then the string is in the language, $\{a^n b^n c^n : n \geq 0\}$, and hence TM returns 'Yes'

If at any stage, the TM does not have any move, it indicates that the string w is not in $\{a^n b^n c^n : n \geq 0\}$, and hence TM returns 'No' indicating w is in $\Sigma^* \sim L$.

This complete the construction of the required TM

2.9 FURTHER READINGS

1. H.R. Lewis & C.H.Papadimitriou: *Elements of the Theory of computation*, PHI, (1981)
2. J.E. Hopcroft, R.Motwani & J.D.Ullman: *Introduction to Automata Theory, Languages, and Computation* (II Ed.) Pearson Education Asia (2001)
3. J.E. Hopcroft and J.D. Ullman: *Introduction to Automata Theory, Language, and Computation*, Narosa Publishing House (1987)
4. J.C. Martin: *Introduction to Languages and Theory of Computation*, Tata-Mc Graw-Hill (1997)

UNIT 3 RECURSIVE FUNCTION THEORY

Structure	Page Nos.
3.0 Introduction	92
3.1 Objectives	93
3.2 Some Recursive Definitions	94
3.3 Partial, Total and Constant Functions	95
3.4 Primitive Recursive Functions	99
3.5 Intuitive Introduction to Primitive Recursion	102
3.6 Primitive Recursion is Weak Technique	112
3.7 The Techniques of unbounded Minimalisation, Partial Recursion and μ -Recursion	115
3.8 Summary	120
3.9 Solutions/Answers	121
3.10 Further Readings	123

3.0 INTRODUCTION

Let us stop for a moment and know that there really is another way (rather, more ways) of looking formally at the notion of computation.*

In the previous units, we have discussed the automata or machine models of the computational phenomenon. The automata approach to computation is *Operational* in nature, i.e., automata approach is concerned with the computational aspect of 'how the computation is to be performed'.

In this unit, we will be concerned with *Recursive Function Theory*, which is a **functional** or **declarative** approach to computation. Under this approach, *computation* is described in terms of 'what is to be accomplished' in stead of 'how to accomplish'.

Each computational theory (rather each theory about any other phenomenon also) starts with some *assumptions*, for example, *about basic (undefined) concepts, operational capabilities and a set of statements, called axioms and postulates, which are assumed to be fundamentally true* (i.e. assumed to be true without any argument). In Automata Theory, the concepts like 'state' 'initial state', 'final state' and 'input' etc are assumed to be understood, without any elaboration. Further, *the capabilities of an automata to accept an input from the environment; to change its state on some, or even on no input; to give signal about acceptability/unacceptability of a string; are assumed.*

In Recursive Function Theory, *to begin with*, it is assumed that **three types of functions** (viz ξ , σ and \prod_k^* which are called *initial functions* and are described under Notations below) and **three structuring rules** (viz combination, composition and primitive recursion) for constructing more complex functions out of the already constructed or assumed to be constructible functions are so simple that *our ability* to construct machines to realize these functions and the structuring rules is taken as *acceptable without any argument*. The functions, obtained by applying a finite sequence of the structuring rules to the initial functions, are called **Primitive Recursive functions**. However, with these simple functions and elementary

structuring rules, though it is possible to construct very complex functions yet, even some simple functions like **division** are not constructible by the above mechanism.

* Two other well-known formalisms are (i) Church's λ -Calculus and (ii) Curry's Combinatory Logic

Therefore, another structuring rule, viz **unbounded minimalization** is added which leads to the concepts of μ -recursion and partial recursion.

Constructibility/Computability has been a pursuit of the mathematicians, since at least the peak of Greek civilization in third/fourth century B.C. The intellectual concern was about the **constructibility of real numbers**, i.e, for a given real number α , to attempt to draw a line of length α , with the help of only an unmarked straight edge and a compass, provided fundamental unit length is given. These attempts at constructibility of real numbers, lead to some *famous problems** including the problems of

(i) *Trisecting an angle*, (ii) *Duplicating a cube* and (iii) *squaring a circle*.

In this unit, the concept of *constructible* or *computable*, the latter being the more often used term in Computer Science, is based only on our *intuitive* understanding of the concept. Discussion of *computable* in the *formal* sense based on *Church-Turing hypothesis*, is taken up in other units.

To some of the learners, the treatment of some of the topics may appear to be undesirably too detailed. However, the details are justified in view of the fact that the subject matter is presented from the point of view of the undergraduate students, many of whom may not have studied Mathematics even at 10+2 level.

In order to facilitate faster coverage of the material by advanced learners, some of the contents are placed in boxes which, without any loss of continuity, can be skipped after first reading or even after a cursory glance.

Note: Exercises in the Block are numbered in one sequence; all other numbered items like theorems, examples, lemmas, statements are taken together for another numbering sequence.

Key words: recursive definition, partial function, total function, initial functions, structuring rules, primitive recursion, bounded minimalization unbounded minimalisation, partial recursion, μ -recursion.

Notations:

- \mathbb{N} : the set of natural numbers including 0
- \mathbb{I} : the set of integers
- ξ : the zero function which maps every element of the domain to 0.
- σ : the successor function, which maps each natural number n to $n + 1$
- \prod_i^k : the projection function which maps the k -tuple $(m_1, \dots, m_i, \dots, m_k)$ to the i th component m_i , for $1 \leq i \leq k$.
- \neg : negation
- \exists : there exists

3.1 OBJECTIVES

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

* For more details refer pp 297-299, *A First Course in Abstract Algebra* by J.B. Fraleigh, VII edition, Pearson Education, 2003.

- To explain the concepts of primitive recursion, μ -recursion and partial recursion
- alongwith other auxiliary concepts
- to tell the hierarchy between the classes of primitive recursive functions, total computable functions, μ -recursive functions and partial recursive functions.
- use these concepts and techniques for generating functions of these classes

3.2 SOME RECURSIVE DEFINITIONS

We are familiar with the concept of factorial of a natural number n , denoted as $n!$, with one of the ways of defining it as:

$$n! = n \cdot (n - 1) \cdot \dots \cdot 1 \tag{1}$$

This is an *explicit* definition of $n!$.

However, the following is an *implicit* definition, called **recursive definition**, of *factorial*.

$$\begin{aligned} 0! &= 1 \quad \text{and} \\ n! &= n \cdot (n - 1)! \quad \text{for } n \geq 1. \end{aligned} \tag{2}$$

The definition (2) above of the factorial is *recursive* in the sense that in order to find the value of factorial at an argument n , we need to find the value of factorial at some simpler argument, in this case $(n-1)$, alongwith possibly some other calculations. In both the explicit and implicit definitions (1) and (2) above of $n!$, our approach is *functional or declarative* in nature, where computation is described in terms of 'what is to be accomplished' instead of 'how to accomplish'.

Similarly, for a natural number n or a real number (or even a complex number) x , the exponential x^n is explicitly defined as

$$x^n = \underbrace{x \cdot x \cdot \dots \cdot x}_{n \text{ times}} \tag{3}$$

Also, the exponential x^n is recursively defined as:

$$\begin{aligned} x^0 &= 1 \\ x^n &= x \cdot x^{n-1}, \text{ for a natural number } n \geq 1. \end{aligned} \tag{4}$$

Remark 1

- We may observe that non-recursive definitions (1) and (3) given above respectively for $n!$ and x^n use the *imprecise* notation ' \dots '. On the other hand, the corresponding recursive definitions (2) and (4) use only *precise* notations.
- In (2) and (4), the definitions are given in terms of their own partial definitions viz. $n!$ in terms of $(n - 1)!$ and x^n in terms of x^{n-1} . In this way, the problem of evaluating $n!$ is *converted* to the problem of evaluation of $(n - 1)!$. This *conversion* of a problem to a less complex version of the problem may be called *reduction* in case we are able to show that calculating $(n - 1)!$ is relatively less complex than calculating $n!$. If we look back on definition (2) of $n!$, we observe that $0!$ is given as a definite number requiring no more applications of the definition of factorial to another number. And reaching $0!$ from $(n - 1)!$ takes lesser number of applications of (2) than reaching $0!$ from $n!$. Thus, we can see that the problem of calculating $n!$ is *reduced* through successive applications of the definition of factorial as given by (2)

and is *terminated* when $0!$ is replaced by 1. Exactly on the similar lines, the problem of calculating x^n is gradually *reduced* by the application of definition (4) and is terminated when x^0 is replaced by 1.

3.3 PARTIAL, TOTAL AND CONSTANT FUNCTIONS

As mentioned under Remarks (ii) above, the factorial of n is defined in terms of *only* the factorial of another, but smaller, number. However, this idea of defining a function in terms of *only* itself may be further generalized when a function f may be defined, in addition to in terms of f itself, possibly in terms of some other functions also. Another way in which the idea of recursion as explained above is generalized, is through extending the scope of recursive definitions to *partial* functions (*to be defined*). Various generalizations, including the one given below, lead to the definitions of *primitive recursion* and *partial recursion*.

The idea of functions from N to N , can be generalized to functions from N^k to N^p where
 $k = 0, 1, 2, \dots$
 $p = 0, 1, 2, \dots$

Example 2: of Functions from $N^k \rightarrow N^p$ where $k > 1$

$$\begin{aligned} \text{Plus: } N \times N &\rightarrow N, \text{ with} \\ \text{plus}(n, m) &= n + m, \end{aligned} \quad (5)$$

Mapping every pair of integers of N to integers in N .
 E.g., *Plus* takes the ordered pair (3, 2) and returns 5. Similarly, *Plus* takes the ordered pair (4, 0) and returns 4.

Similarly, we may define

$$\begin{aligned} \text{PROD: } N \times N &\rightarrow N, \text{ with} \\ \text{PROD}(m, n) &= m \cdot n \text{ for } m, n \in N. \end{aligned} \quad (6)$$

And we may define

$$\text{Exp}(m, n) = m^n \text{ for all } m, n \in N \quad (7)$$

Example 3: of a function from N^k to N^p where $k > 1$ and $p > 1$:

$$\begin{aligned} \text{Plus-Prod: } N^2 &\rightarrow N^2, \text{ such that} \\ \text{Plus-Prod}(m, n) &= (m + n, m \cdot n) = (\text{Plus}(m, n), \text{Prod}(m, n)) \end{aligned} \quad (8)$$

In other words, the function **Plus-Prod** takes a pair of elements m and n of N and maps this pair (m, n) to a pair of integers, viz, $(m + n)$ and $(m \cdot n)$

Also, we may define the function

$$\begin{aligned} \text{Plus-Prod-Exp: } N^2 &\rightarrow N^3 \text{ with} \\ \text{Plus-Prod-Exp}(m, n) &= (m + n, m \cdot n, m^n) \\ &= (\text{Plus}(m, n), \text{Prod}(m, n), \text{Exp}(m, n)) \end{aligned} \quad (9)$$

Here the ordered pair (m, n) is mapped to the ordered triple of three integers, viz, $(m + n)$, $(m \cdot n)$ and m^n

Example 4: of function from $N^k \rightarrow N^q \times N^p$ where $k, q, p \in N$

A somewhat similar but distinct function say

New-Plus-Prod-Exp: $N^2 \rightarrow N^2 \times N$
 may be defined as

$$\begin{aligned} \text{New-Plus-Prod-Exp}(m, n) &= ((m+n, m \cdot n), m^n) \\ &= (\text{Plus-Prod}(m, n), \text{Exp}(m, n)) \end{aligned} \quad (10)$$

Please note the minute difference between Plus-Prod-Exp and New-Plus-Prod-Exp

Remarks 5

In the definitions under (5) to (10) above, among other facts, we may observe that earlier defined functions may be used in defining more complex functions. Our ability to define more and more complex functions in terms of earlier defined functions, plays a very important role in the study of primitive recursion and partial recursion etc, which are generalizations of the concept of recursion discussed in defining $n!$ and x^n etc.

The recursive definitions of Plus, Prod etc. will be discussed later.

The constant Functions:

Though it is not intuitive, yet we may have functions on N which do not require any argument.

Consider the function

$$\begin{aligned} C_5 : N &\rightarrow N \text{ such that} \\ C_5(n) &= 5, \text{ for all } n \in N \end{aligned} \tag{11}$$

In view of the fact that the value 5 is independent of n in (11), we can very well write (11) as

$$C_5() = 5, \tag{12}$$

Given the fact that we are considering domains of functions as N^k for $k \in N$, we extend our notation for functions from $N^k \rightarrow N$ to include functions from $N^0 \rightarrow N$, and rewrite (11) as

$$\begin{aligned} C_5 : N^0 &\rightarrow N \text{ such that} \\ C_5() &= 5. \end{aligned}$$

Also, in order to include in the notation itself the fact that the function takes zero number of arguments, we may use the notation C^0 instead of C , i.e.,

$$C^0_5 = 5 \tag{13}$$

Generalizing the constant function C^0_5 , we may define

$$\begin{aligned} C^0_q : N^0 &\rightarrow N \text{ such that} \\ C^0_q() &= q, \text{ for some fixed integer } q \text{ in } N. \end{aligned}$$

Further, we can extend the set of constant functions to include the functions

$$\begin{aligned} C^k_q : N^k &\rightarrow N \text{ such that} \\ C^k_q(n_1, n_2, \dots, n_k) &= q, \\ \text{for } n_1, n_2, \dots, n_k \in N \text{ and for some integers } k \text{ and } q \text{ in } N. \end{aligned} \tag{14}$$

Partial Function

We are already familiar with the concept of *function* in the mathematical sense. Informally, for two given sets X and Y a **function**

$$f : X \rightarrow Y$$

is a **rule** that associates to **each** element x of X a **unique** element y of Y . Here X is called the *domain* of the function f and Y the *codomain* of f . (15)

However, in order to extend the class of computable functions beyond the class of primitive recursive functions (*to be defined*), to partial-recursive functions (*to be*

defined), we relax the condition 'for each element x of X ' in the definition of function leading to the following definition.

Partial Function: A partial function is a rule

$$f: X \rightarrow Y$$

that associates elements of Y to elements of X in such a way that, for $y_1 \in Y$ if there exists an element x_1 of X s.t. $f(x_1) = y_1$, then there is no element y_2 of Y , with $y_1 \neq y_2$, s.t. $f(x_1) = y_2$. (16)

However, there may be some elements x of X for which there may not be any y such that $f(x) = y$. In other words, in the definition of a *partial* function, it is *not necessary* that for each element x of X , there *must* be an element y of Y that corresponds to x under f . However, for an element x of X , if there is an element y_1 of Y that corresponds to x under f , then there *can not* be a y_2 in Y with $y_1 \neq y_2$ such that y_2 also corresponds to x under the partial function under consideration.

Example 6: We consider a rule of correspondance *Quot*: $N \times N \rightarrow N$ that takes a pair (m, n) of integers and associates an integer q , if it exists, s.t. $m = nq + r$ with $0 \leq r < n$. Now if $n=0$ then no r with $0 \leq r < 0$ exists implying *Quot* (m, n) is not defined for $n=0$. Thus *Quot* is a *partial* function, but not a *function* or a *total* function as is going to be defined below.

Total Function: If a partial function satisfies the condition given in (15), i.e., it is a function in the conventional sense, then it will be called **Total Function**. The adjective *total* is added to a function in conventional sense in order to differentiate the function in conventional sense from the *partial* function which satisfy condition (16) but do not satisfy the condition (15) above.

Remarks 7

In this block, unless it is mentioned otherwise we will be dealing with functions (partial or total), the domains of which are only of the form $N^k = N \times \dots \times N$ for $k \in N$.

Functions with domain N^k are called *k-place functions*.

Also, unless it is mentioned otherwise, the functions under consideration are restricted to the ones that have N as their **codomain**. Our consideration of only the functions of the form $f: N^k \rightarrow N$, is not a major restriction, because using some encoding techniques like *Gödel Numbering*, any domain can be expressed as a subset of the set N^k .

Why we need partial functions?

We know there are infinitely many possible sets which can be represented by finite means. For example the infinite set N is **finitely representable by the following two statements:**

- (i) 0 is a member of N and
- (ii) if n is a member of N then so is $\sigma(n)$ (i.e., $(n+1)$).

And for each such non-empty set X , at least one function, say the identity function $I: X \rightarrow X$, can be defined. Also, for such sets X, Y, Z etc., we can think of new sets which may be the product sets, for example $X \times Y, Y \times Z \times X$, just to name a few. Each of these product sets itself can be the domain (or even the range) of some functions.

Thus, unless we use an ingenious method of the type described below, the general discussion of functions would involve consideration of *infinitely many types* of domains and codomains, even if, each of these may be finitely representable.

By an appropriate encoding, it can be easily seen that each countable set can be thought of as either N^k or as a proper subset of N^k , for some $k = 0, 1, 2, \dots$

For example

The set $X = \{a, b, c, \dots, z\}$ can easily be thought of as a subset of N , by using the following encoding

$a \rightarrow 1$

$b \rightarrow 2$

\vdots

\vdots

\vdots

$z \rightarrow 26$

Thus, functions, instead of being considered between arbitrary but countable domains and codomains, may be considered as functions of the form $P \rightarrow N^m$, where P either equals an N^k or is a proper subset of N^k , for suitable integers k and m .

However, any function $f: X \rightarrow N^m$ for $X = \{a, b, \dots, z\}$ above when considered after an encoding as a function $f: N \rightarrow N^m$, **cannot be total**, because $f(m)$ for $m \geq 27$, is not defined. In general, any function with a *finite* domain when considered as a function between the encoded sets N^k and N^m must be *strictly partial*.

Also, for large number of functions involved in the solution of everyday problems, each has a finite domain.

Thus, in order to simplify the discussion of functions with arbitrary but countable domains and ranges, it is possible, through appropriate encoding, to consider such a function as a function of the form $N^k \rightarrow N^m$ provided functions are allowed to be partially defined on the domain N^k .

Examples 8: of total and Partial Functions

(i) **The successor function** $S: N \rightarrow N$, s.t., $S(x) = x + 1$ for all $x \in N$. The function **S is a total function from N to N** . Successor function plays an important role in the recursion theory. Therefore, it is useful to know that even the notation σ is used to denote the successor function.

(ii) The function
Plus: $N^2 \rightarrow N$ such that for all $x, y \in N$.
 Plus $(x, y) = x + y$,
 is also a *total* function

(iii) However, the function
Minus: $N^2 \rightarrow N$ such that
 Minus $(x, y) = x - y$ for $x \geq y$ in N ,
 is only a *partial* function, which is *not* a total function.
 In other words, *Minus* is a *strictly partial* function.

However, a slightly different function say **Minus_Int** becomes total, if we allow

Minus_Int: $N^2 \rightarrow I$,

with I , the set of integers as codomain and the rule of correspondance given by
Minus_Int $(x, y) = x - y$ for $x, y \in N$. (in stead of, for just $x \geq y$)

[†] While talking of total or partial functions, it is understood that the domain is of the form N^k .

However, as mentioned earlier, we are restricting to only functions of the form $f: \mathbb{N}^k \rightarrow \mathbb{N}$.

Therefore, if required, we discuss only the *strictly partial* function *Minus*.

However, we will discuss a scheme of discussing *Minus-Int* as a function from \mathbb{N}^2 to \mathbb{N}^2 .

- (iv) corresponding to the partial function *minus*, there is a well-known function **monus**, also denoted as $\dot{-}$ and defined as

$$\text{Monus}(x, y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases}$$

Monus (x, y) may also be written as $x \dot{-} y$.

- (v) We define the function *div* from \mathbb{N}^2 to \mathbb{N} with $\text{div}(x, y) = z$, only for those pairs (x, y) of elements of \mathbb{N} for which $x = y \cdot z$ for some z from \mathbb{N} . Then *div* is strictly partial.
- (vi) The **Square-Root function**, named as say **SQRT**, (also denoted by $\sqrt{\quad}$) and given by
 SQRT: $\mathbb{N} \rightarrow \mathbb{N}$
 such that for $x, y \in \mathbb{N}$,
 SQRT $(x) = y$ if $y^2 = x$.

Again SQRT is a *strictly partial function*.

After having provided the necessary background, we explain the concept of **primitive recursive functions** the set of which forms a proper subset of the set of total functions. As the discussion of general **partial recursive functions** requires introduction of some more background material, the general partial recursive functions will be discussed later.

We mentioned earlier that, **each of the approaches to computation starts with some elementary entities** of some domain and some **structuring rules**, where the rules are easily applicable to form more and more complex entities of the domain.

3.4 PRIMITIVE RECURSIVE FUNCTIONS

The set of primitive recursive functions is obtained by three types of **initial functions** (which are *elementary* primitive functions) and three **structuring rules** for constructing more complex functions from already constructed functions.

Three types of initial functions are

- (i) The 0-Place *zero function* ξ from \mathbb{N}^0 to \mathbb{N} such that $\xi(\) = 0$ (19)
- (ii) The *successor function*
 $\sigma: \mathbb{N} \rightarrow \mathbb{N}$
 such that
 $\sigma(n) = n + 1$ for all $n \in \mathbb{N}$. (20)

* In the literature, two of the three structuring rules are combined in one rule and hence, in most of the literature, number of structuring rules is mentioned as TWO and not THREE. However, then presentation of the subject matter becomes too complex from the point of view of undergraduate students.

(iii) *The Projections:* We know that for $k \geq 1$, N^k is the set of all k -tuples of the form $\bar{n} = (n_1, n_2, \dots, n_i, \dots, n_k)$ for $1 \leq i \leq k$.

For each fixed i , with $1 \leq i \leq k$, we may define a function, denoted by \prod_i^k , with

$$\begin{aligned} \prod_i^k : N^k &\rightarrow N \text{ such that for } \bar{n} = (n_1, n_2, \dots, n_i, \dots, n_k) \\ \prod_i^k \bar{n} &= \prod_i^k (n_1, n_2, \dots, n_i, \dots, n_k) \\ &= \text{ith component of } (n_1, n_2, \dots, n_i, \dots, n_k) = n_i \end{aligned} \quad (21)$$

Thus, we have defined k projection functions, each with domain N^k , viz.

$\prod_1^k, \prod_2^k, \dots, \prod_i^k, \dots, \prod_k^k$ and each of which maps to N .

For the sake of explanation, we have $\prod_3^5 (2, -7, 12, 4, 3) = 12$

Finally, the zero-place zero function ξ , the successor function σ and the projection function \prod_i^k for $k \in N$, and $i \in N$, with $1 \leq i \leq k$, are the *only* initial functions, which are also called *elementary primitive functions*.

Ex.1) Prove that each of the elementary primitive function viz zero function ξ , successor function σ and each of the projection functions \prod_i^k , $1 \leq i \leq k$, is a *total* function.

In the very beginning itself, it was mentioned that *computability of functions* is a major concern in the Theory of Computation.

In this context, consider the following

Statement 9: The initial functions ξ, σ and \prod_i^k are all computable*.

The statement is not a *theorem*, the truth of which can be established through a *proof*. The statement is axiomatic in the sense that it should not only be intuitively correct but should be fundamentally true in the sense that it can not be otherwise. In spite of the above, we give an informal argument in support of the apparent truth of the statement. The *computability of the initial function ξ* is about our capability of constructing a machine to perform the activity of writing the symbol 0. This capability can be assumed without any doubt. Similarly, our capability of constructing a machine which returns $(n+1)$ for each input n , can also be assumed without any doubt. Hence, we may assume that the successor function σ is computable.

Finally, *computability of a projection function* say \prod_i^k , is about the designing of a machine capable of scanning a k -type say $\bar{m} = (m_1, m_2, \dots, m_i, \dots, m_k)$ starting with the first component m_1 and go on moving to the right till i th component m_i is scanned and then writing m_i as output. In view of the type of machines available, it can be safely assumed that we can construct a machine to execute these activities required for a projection function.

Next, we define the three structuring rules which are known as

(i) **combination**

* Computability in the formal sense of Church-Turing Thesis has been discussed in other unit. Here computability is taken as an intuitive informal notion.

- (ii) composition and
- (iii) primitive recursion.

Remarks 10

Before providing the definitions for the above-mentioned structuring rules, it may be stated that by applying these structuring rules, to begin with, to the initial functions and then by successive applications of these structuring rules to the functions already obtained by previous applications, we can construct quite complex functions

(I) Combination as a structuring rule

The combination of two functions

$$g: N^k \rightarrow N$$

$$h: N^k \rightarrow N$$

is a function

$$f: N^k \rightarrow N \times N$$

such that for $(n_1, \dots, n_k) = \bar{n} \in N^k$,

$$f(\bar{n}) = (g(\bar{n}), h(\bar{n})). \tag{22}$$

Then, the function f is denoted by $g \times h$ and is called combination of g and h .

Remarks 11

In stead of $g: N^k \rightarrow N$ and $h: N^k \rightarrow N$, we may take $g: N^k \rightarrow N^p$ and $h: N^k \rightarrow N^q$, and then we get a function $f: N^k \rightarrow N^{p+q}$ from the definition of f given by (22).

(II) Composition as a structuring Rule

Let

$$g: N^k \rightarrow N^p$$

and

$$h: N^p \rightarrow N^q$$

for $k, p, q \in N$

be two given functions.

Then we define a function

$$f: N^k \rightarrow N^q$$

as follows:

if $(n_1, n_2, \dots, n_k) \in N^k$ and

$$g(n_1, n_2, \dots, n_k) = (m_1, m_2, \dots, m_p) \in N^p$$

Further, if

$$h(m_1, m_2, \dots, m_p) = (t_1, t_2, \dots, t_q) \in N^q$$

then

$f: N^k \rightarrow N^q$ is such that

$$f(n_1, n_2, \dots, n_k)$$

$$= h(g(n_1, n_2, \dots, n_k))$$

$$= h(m_1, m_2, \dots, m_p) = (t_1, t_2, \dots, t_q) \tag{23}$$

The function f , so defined, is called the composition of g and h and is denoted by $h \circ g$ (Some authors use the notation $g \cdot h$ instead)

Examples 12:

(i) If $g: N \rightarrow N$ is given by

$$g(n) = n^2 \quad \text{and}$$

$h: N \rightarrow N$ is given by

$$h(n) = (n + 3)$$

Then we define a function

$$f: N \rightarrow N$$

such that

$$f = h \circ g, \text{ then}$$

$$f(n) = h(g(n)) = h(n^2) = n^2 + 3 \text{ for all } n \in N$$

- (ii) Functions g and h are as given in Example 12.(i) above. Let us define a function $k: \mathbb{N} \rightarrow \mathbb{N}$ such that $k = g \cdot h$, then
 $k(n) = g(h(n))$
 $= g(n+3) = (n+3)^2$
 for all $n \in \mathbb{N}$

Ex. 2) What is the result of applying the function $(\prod_2^2 \times \prod_1^2) \cdot (\prod_1^4 \times \prod_1^4)$ to four tuple $(8,7,4,2)$?, where $(f \times g)(x) = (f(x), g(x))$

Ex. 3) A function $f: \mathbb{N} \rightarrow \mathbb{N}$ is defined as
 $f(0) = \xi(\)$ and
 $f(y) = \sigma \cdot \sigma \cdot \sigma \cdot f(y-1)$
 What is the value of $f(4)$?

Remarks 13

As mentioned earlier the two structuring rules discussed so far, viz, combination rule and composition rule, are presented in the literature as a single rule, and is generally called as composition rule, which is actually a generalization of our composition and combination rules. We call this rule as **Generalised Composition Rule** and discuss it below:

Let the function

$$g: \mathbb{N}^k \rightarrow \mathbb{N}^p \quad \text{and}$$

$$h_1, h_2, \dots, h_k \text{ are } k \text{ functions with}$$

$$h_i: \mathbb{N}^m \rightarrow \mathbb{N}, \text{ for } i = 1, 2, \dots, k$$

Then we define a function

$$f: \mathbb{N}^m \rightarrow \mathbb{N}^p \text{ such that for } \bar{n} = (n_1, \dots, n_m) \in \mathbb{N}^m$$

$$f(\bar{n}) = g(h_1(\bar{n}), h_2(\bar{n}), \dots, h_k(\bar{n})). \tag{24}$$

Then f is said to be obtained from g, h_1, h_2, \dots, h_k by *generalized composition* or if there is no confusion just by *composition*.

Ex.4) Show that the *combination* rule given by (22) and *composition* rule given by (23) are special cases of the *generalized composition* rule given by (24).

Next, we consider the *third structuring rule*, viz *Primitive Recursion*. As the rule requires comprehensive discussion, we discuss it in an independent section below.

3.5 INTUITIVE INTRODUCTION TO PRIMITIVE RECURSION

Earlier, we considered the recursive definition of $n!$ for $n \in \mathbb{N}$ as:
 $0! = 1$ and
 $n! = n \cdot ((n-1)!) \quad \text{for } n \geq 1.$

Also, we considered for recursive definition of the exponential x^n of x , a real number and $n \in \mathbb{N}$ as:

$$x^0 = 1 \quad \text{and}$$

$$x^n = x \cdot x^{n-1} \quad \text{for } n \geq 1.$$

In order to understand the generalization of the recursive definitions considered above, let us consider the following example.

Example 14: As an Intuitive Introduction to Primitive Recursion.

Let us consider a special kind of tree that initially has only one branch, which is treated as a *new* branch. At the end of each year, out of each new branch, m new branches grow out (we call m as the **branching factor of the tree**). And the branch, out of which branches grew out once, is no more a new branch. We define a *function* say f which gives the total number of branches in the tree after n years, assuming the branching out process continues for ever (or at least for more than n years) at the rate of m branches per year.

It is clear that the function f depends on both m and n . In order to facilitate the understanding of the process of getting f as a function of m and n , let us initially consider m as a constant. Also to begin with, we consider only the function $b(n)$ that returns the number of new branches that are generated at the end of the n th year for:

$n = 1, 2, \dots$. Subsequently, $b(n)$ shall be used in computing $f(m, n)$, where

- $b(1)$ = After one year, number of new branches = m
- $b(2)$ = After two years, number of new branches = $m \cdot b(1) = m \cdot m = m^2$
- $b(3)$ = After three years, number of new branches = m (number of new branches after 2 years) = m^3 .

Continuing like this, we get,

$$\begin{aligned} b(n) &= \text{After } n \text{ years, number of new branches} \\ &= m(\text{number of new branches after } (n-1) \text{ years}) \\ &= m \cdot b(n-1) = mm^{n-1} = m^n \end{aligned}$$

Next, we consider $f(m, n)$, the number of all branches at the end of n years.

$$\begin{aligned} f(m, 1) &= \text{Total Number of branches after one year} \\ &= \text{old branches at the end of one year} + \text{new branches generated at the} \\ &\quad \text{end of the first year, i.e.,} \\ f(m, 1) &= 1 + m \\ f(m, 2) &= \text{Total number of branches after two years} \\ &= \text{Old branches at completion of two years} + \text{New branches generated} \\ &\quad \text{at the end of second year, i.e.,} \\ f(m, 2) &= f(m, 1) + m^2 \\ f(m, n) &= \text{Continuing like this, total number of branches after } n \text{ years} \\ &= \text{Total number of branches after } (n-1) \text{ year} + \text{New branches at the end} \\ &\quad \text{of the } n \text{th year, i.e.,} \\ &= f(m, n-1) + m \cdot b(n-1), \text{ i.e.,} \\ f(m, n) &= f(m, n-1) + m^n \end{aligned} \tag{25}$$

For a short while, if we denote $f(m, n-1)$ as t

Then (25) becomes

$$\begin{aligned} f(m, n) &= t + m^n, \quad \text{a function of } m, n \text{ and } t, \text{ say} \\ &= h(m, n, t), \quad \text{for some function } h: N^3 \rightarrow N. \end{aligned}$$

Summarising, we get

$$\begin{aligned} f(m, n) &= h(m, n, t), \quad \text{for some function } h: N^3 \rightarrow N. \\ &\quad \text{Replacing } t \text{ by } f(m, n-1), \text{ we get} \\ f(m, n) &= h(m, n, f(m, n-1)) \end{aligned}$$

Thus $f(m, n)$, the number of all branches immediately on completion of n years can be defined as:

$$\begin{aligned} f(m, 0) &= 1 = m^0 \\ f(m, n) &= h(m, n, f(m, n-1)) \end{aligned} \tag{26}$$

where

$$h(m, n, t) = m^n + 1$$

In order to further generalize the concept of recursion, we consider the same problem with a little difference by taking a new starting time (i.e., zeroth year) for the problem of counting of the number of branches when the tree already has, say, $1 + m + m^2 + m^3$ branches. (i.e., the initial branch is already 3 years old) and let $J(m, n)$ denote the number of total branches after n years of the new starting time, where m is the branching factor.

Then $J(m, n)$ is defined as:

$$J(m, 0) = 1 + m + m^2 + m^3, \text{ which is some function say } g(m) \text{ of } m, \text{ i.e.,}$$

$$J(m, 0) = g(m) \quad \text{and}$$

$$J(m, n+1) = J(m, n) + m^{n+4},$$

which is some function say L of m, n and $J(m, n)$, i.e.

$$J(m, n+1) = L(m, n, J(m, n)).$$

Summarizing, the function $J(m, n)$ may be defined as

$$J(m, 0) = g(m) \quad \text{and} \quad (27)$$

$$J(m, n+1) = L(m, n, J(m, n)), \quad (28)$$

for some function g of m and another function L of m, n and $J(m, n)$.

The above discussion can be generalized still further when instead of one tree, initially, we have k trees T_1, T_2, \dots, T_k , s.t. for the i th tree T_i , the branching factor is m_i for $i = 1, 2, \dots, k$.

Also, we assume that different trees started growing (i.e., having their first branches) in different calendar years. After all these trees have started growing, some calendar year is taken as starting or zeroth year for the purpose of counting the number of branches in all the trees taken together.

Then (27) and (28) can be rewritten as

$$J_i(m_i, 0) = g_i(m_i)$$

$$J_i(m_i, n+1) = L_i(m_i, n, J_i(m_i, n)) \quad \text{for } i = 1, 2, \dots, k. \quad (29)$$

(where $g_i(m_i)$ denote the number of branches, in the zeroth year, of the i th tree whose branching factor is m_i)

Then total number of branches on all the k trees, after completion of n years after the starting year, may be defined by a function F of m_1, m_2, \dots, m_k and n as follows:

$$F(m_1, m_2, \dots, m_k, 0) = g_1(m_1) + g_2(m_2) + \dots + g_k(m_k) = G(m_1, m_2, \dots, m_k),$$

for some function G of m_1, m_2, \dots, m_k and

$$F(m_1, m_2, \dots, m_k, n+1) = J_1(m_1, n+1) + J_2(m_2, n+1) + \dots + J_k(m_k, n+1) \quad (30)$$

But by (29), each $J_i(m_i, n+1)$ is a function of $J_i(m_i, n)$

$$\text{If } \bar{m} = (m_1, m_2, \dots, m_i, \dots, m_k)$$

Then in view of these facts, i.e.,

(i) $F(\bar{m}, n+1)$ is a sum of $J_i(m_i, n+1)$ by (30)

(ii) each $J_i(m_i, n+1)$ is a function of $J_i(m_i, n)$ in addition to being a function of m_i and n by (29)

(iii) the sum of $J_i(m_i, n)$, $1 \leq i \leq k$, is an expression which can be obtained by replacing $n+1$ by n in the R.H.S of (30) and hence, by replacing $(n+1)$ by n in the L.H.S of the equality (30),

this sum of $J_i(m_i, n)$'s must be of the form $F(\bar{m}, n)$

In view of (i), (ii) & (iii) above, $F(\bar{m}, n + 1)$ is some function H of $F(\bar{m}, n)$ in addition to being function of m_i 's and n

Then the above definition of F may be rewritten as

$$\begin{aligned} F(\bar{m}, 0) &= G(\bar{m}) && \text{and} \\ F(\bar{m}, n + 1) &= H(\bar{m}, n, F(\bar{m}, n)) \end{aligned} \quad (31)$$

This is exactly what, in formal sense, we say that F is obtained from G and H by primitive recursion. Restating the above, we get the formal

Definition: Primitive Recursion

For $k \geq 0$, a function

$$f: N^{k+1} \rightarrow N^m$$

is said to be constructed using primitive recursion from the functions

$$\begin{aligned} g: N^k &\rightarrow N^m && \text{and} \\ h: N^{k+m+1} &\rightarrow N^m, \end{aligned}$$

if, for $\bar{x} \in N^k$ and $y \in N$,

$$\begin{aligned} f(\bar{x}, 0) &= g(\bar{x}) && \text{and} \\ f(\bar{x}, y + 1) &= h(\bar{x}, y, f(\bar{x}, y)), \end{aligned} \quad (32)$$

The above discussion about three *initial functions* viz. the zero-place *zero function* $\xi()$, the *successor function* σ and the projections \prod_i^k and about the three *structuring rules* viz. *combination*, *composition* given by the discussion preceding and including (22), (23) and *primitive recursion* given by (31), leads to the following **definition**:

Primitive Recursive Function: A function f is primitive recursive function if (and only if) either

(i) it is one of the initial functions viz. $\xi()$, σ or one of the projections

$$\prod_i^k, i \leq k, \text{ or}$$

(ii) it is obtained by application of some finite sequence of structuring rules viz combination, composition, and primitive recursion to the initial functions.

Remark 15

It is implied in the above definition that if a function f is obtained by a finite sequence of application of structuring rules including primitive recursion to some functions (*not necessarily initial functions*) say g_1, g_2, \dots, g_k , each of which has been obtained earlier by application of some finite sequence of combination, composition and *primitive recursion* to some of the *initial functions*, then F must be *primitive recursive*. The implication follows from the fact that F can be obtained from initial functions by first applying sequences of combination, composition and primitive recursion to obtain each of g_1, g_2, \dots, g_k , and then applying a sequence of combination, composition and primitive recursion to get f from g_1, g_2, \dots, g_k .

Examples 16: All functions considered below are from N^k to N , for some $k \in N$.

Example 16(i): The well-known binary function *plus* $(m, n) = m + n$ is primitive recursive, because

$$\begin{aligned} \text{Plus}(m, 0) &= \prod_1^1(m) \\ \text{Plus}(m, n + 1) &= \sigma \cdot \prod_3^3(m, n, \text{plus}(m, n)), \text{ for } n \geq 0 \end{aligned} \quad (33)$$

Example 16 (ii): The well-known binary function Product (written here as *prod*) and given by

prod (*m*, *n*) = *m* · *n* is primitive recursive, because

$$\text{prod}(m, 0) = \xi()$$

$$\text{prod}(m, n + 1) = \text{plus} \left(\prod_1^3 (m, n, \text{prod}(m, n)), \prod_3^3 (m, n, \text{prod}(m, n)) \right) \quad (34)$$

As plus has already been shown to be primitive recursive, therefore, prod is also primitive recursive.

Example 16 (iii): In this example we consider simple function say *int-plus* whose domain and codomain are not \mathbb{N} but \mathbb{I} , the set of all integers including negative integers and zero. We show that the function

$$\text{int-plus} : \mathbb{I} \times \mathbb{I} \rightarrow \mathbb{I}$$

is primitive-recursive.

The proof is based on the fact that Each integer can be thought of as a member of \mathbb{N}^2 , for example, 5 may be thought of as (6, 1) and -5 as (1, 6). In general, $(m, n) \in \mathbb{N}^2$ denotes the integer $m - n$.

Then the function *int-plus*: $\mathbb{I} \times \mathbb{I} \rightarrow \mathbb{I}$ can be thought of as

$$\text{int-plus} : (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N} \times \mathbb{N}$$

Such that, if (m_1, n_1) and $(m_2, n_2) \in \mathbb{N} \times \mathbb{N}$ then

$$\text{int-plus}((m_1, n_1), (m_2, n_2)) = (m_1 + m_2, n_1 + n_2)$$

$$= (\text{plus}(m_1, m_2), \text{plus}(n_1, n_2))$$

(35)

Plus is already shown to be primitive recursive and combination of two primitive recursive functions (viz plus and plus) is primitive recursive. Hence the above equation (35) shows that *int-plus* is a primitive recursive function.

Example 16(iv): The factorial function

$$f(n) = n! \quad \text{for } n \in \mathbb{N},$$

is primitive recursive.

The proof follows from the following argument based on Principle of Mathematical Induction:

Base Case:

$$\text{for } n = 0$$

$$f(0) = 0! = 1 = \xi()$$

Induction Hypothesis:

Let $f(p)$ be primitive recursive

Induction step:

$$f(p + 1) = (p + 1)! = (p!) \cdot (p + 1) = f(p) \cdot (p + 1) = \text{prod}(f(p), p + 1).$$

Using Induction Hypothesis and the fact that product is primitive recursive, $f(p + 1)$ is primitive recursive.

Generalizing the above example, we get the

Theorem 17:

Let

$$g: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$$

be primitive recursive.

Then, for an $\bar{m} \in \mathbb{N}$, the function
 $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$
 given by

$$f(\bar{n}, m) = \prod_{i=0}^m g(\bar{n}, i) = g(\bar{n}, 0) \cdot g(\bar{n}, 1) \cdots g(\bar{n}, m) \quad (36)$$

with $\bar{n} = (n_1, n_2, \dots, n_k) \in \mathbb{N}^k$,
 is primitive recursive

Proof:

We prove the result by Principle of Mathematical Induction on m

Base case:

When $m = 0$, by (36), we get

$$f(\bar{n}, 0) = g(\bar{n}, 0).$$

As g is given to be primitive recursive, the base case follows

Induction Hypothesis:

for $m = p$ we assume

$$f(\bar{n}, p) = \prod_{i=0}^p (g(\bar{n}, i))$$

is primitive recursive

Induction Step:

Consider

$$\begin{aligned} f(\bar{n}, p+1) &= \prod_{i=0}^{p+1} (g(\bar{n}, i)) = \prod_{i=0}^p (g(\bar{n}, i)) \cdot g(\bar{n}, p+1) \\ &= f(\bar{n}, p) \cdot g(\bar{n}, p+1) = \text{prod}(f(\bar{n}, p), g(\bar{n}, p+1)). \end{aligned}$$

In view of the Induction Step and the fact that both g and product are primitive recursive, we get $f(\bar{n}, p+1)$ is primitive recursive.

Definition: The function f , given by (36) above, is said to be obtained from g by Bounded Product. Thus the above theorem may be restated as

Theorem 17:

Bounded Product of a primitive function is primitive recursive

Ex.5) Show that each of the following, earlier defined, functions is primitive recursive:

- (i) Plus-Prod (given by equation(8))
- (ii) Exp
- (iii) New-Plus-Prod-Exp (given by equation (10))

Ex. 6) Show that the predecessor function $\text{pred}: \mathbb{N} \rightarrow \mathbb{N}$ defined as

$$\text{pred}(n) = \begin{cases} 0 & \text{if } n = 0 \\ n-1 & \text{if } n \geq 1 \end{cases}$$

is primitive recursive.

We recall the definition of constant functions:

For each $k \geq 0$ and each $j \geq 0$, a constant function C_j^k maps each k -tuple (m_1, m_2, \dots, m_k) to the fixed integer j , i.e.

$$C_j^k: \mathbb{N}^k \rightarrow \mathbb{N} \text{ such that } C_j^k(m_1, m_2, \dots, m_k) = j, \quad \text{for all } (m_1, m_2, \dots, m_k) \in \mathbb{N}^k.$$

We show that C_j^k are all primitive recursive functions. To begin with, consider the

Lemma 18: Each of the functions C_j^0 for $j \geq 0$ is primitive recursive.

Proof: The proof is presented in two parts:

Case (i) C_0^0 is the function which maps a zero-tuple to the constant 0. It is primitive recursive, because

$$C_0^0 = \xi$$

Case (ii) Each of the functions C_j^0 for $j \geq 1$ is primitive recursive

$$\text{As } C_1^0 = 1 = \sigma \cdot \xi,$$

therefore, C_1^0 is primitive recursive

$$\text{Again as } C_2^0 = \sigma \cdot C_1^0$$

and C_1^0 is already shown to primitive recursive, therefore, C_2^0 is primitive recursive.

We use mathematical induction on j to show C_j^0 is primitive recursive for all j .

Base case. For $j = 0$, we have already shown, that C_0^0 is primitive recursive.

Induction Hypothesis: Let C_n^0 is primitive recursive, for any integer n .

Induction step

$C_{n+1}^0 = \sigma \cdot C_n^0$ By induction hypothesis, C_n^0 is assumed to be primitive recursive and σ is primitive recursive and composition of two recursive functions is primitive recursive, therefore, C_{n+1}^0 is primitive recursive.

Hence by Principle of mathematical induction, C_j^0 is primitive recursive, for all $j \in \mathbb{N}$.

The lemma proves C_j^k as primitive recursive only for $k = 0$.

The proof for the general integer k follows from the

Theorem 19: The constant function C_j^k , for $k \geq 0$ and $j \geq 0$, is primitive recursive.

Proof: We prove the theorem by induction on k .

Base case: When $k = 0$, the proof follows from the lemma, in which we proved that C_j^0 is primitive recursive for all j .

Induction Hypothesis: Assume C_j^i is primitive recursive, for all integers j and all integers $i \leq p$.

Induction step:

$$\text{Let } \bar{m} = (m_1, m_2, \dots, m_p) \in \mathbb{N}^p$$

Now $C_j^{p+1}(\bar{m}, 0) = C_j^p(\bar{m})$, each of the two sides of the equality, is equal to j

$$C_j^{p+1}(\bar{m}, n+1) = \prod_{p+2}^{p+2} (\bar{m}, n, C_j^{p+1}(\bar{m}, n))$$

Hence, the theorem is proved.

Let us try the following exercises

Ex. 7) The monus function defined earlier as

$$\text{monus}(m, n) = \begin{cases} m - n, & \text{if } m \geq n \\ 0, & \text{otherwise,} \end{cases}$$

is primitive recursive.

Ex. 8) Show that following function

$$\text{eq}(m, n) = \begin{cases} 1, & \text{if } m = n \text{ and} \\ 0, & \text{else} \end{cases}$$

is primitive recursive.

Ex. 9) Show that the function minus: $I \times I \rightarrow I$, with

$$\text{Minus}(m, n) = m - n \text{ for all } m, n \in I,$$

where I is the set of all integers, is primitive recursive

Ex. 10) Show that the function

$$\neg \text{eq}(m, n) = \begin{cases} 1, & \text{if } m \neq n \\ 0, & \text{if } m = n \end{cases}$$

is primitive recursive

Ex. 11) Show that for $i \in \mathbb{N}$, characteristic functions

$$K_i(n) = \begin{cases} 1, & \text{if } n = i \\ 0, & \text{otherwise} \end{cases}$$

is primitive recursive

Ex. 12) Show that the function

$f: \mathbb{N} \rightarrow \mathbb{N}$ given by

$$f(n) = \begin{cases} 7 & \text{when } n = 0 \\ 12 & \text{when } n = 5 \\ 8 & \text{when otherwise} \end{cases}$$

is primitive recursive.

Statement 20: The structuring rules viz *combination, composition and primitive recursion* produce computable functions from computable functions.

Like Statement 9 earlier, no formal proof of the statement is possible. As earlier, we present an informal/intuitive argument in support the apparent truth of the Statement (20).

(i) **Composition Rule produces computable functions from computable functions**

Let

$$\begin{aligned} g &: \mathbb{N}^k \rightarrow \mathbb{N}^m & \text{and} \\ h &: \mathbb{N}^k \rightarrow \mathbb{N}^m \end{aligned}$$

be computable functions.

Then value $f(\vec{n})$ of $\vec{n} = (n_1, n_2, \dots, n_k) \in \mathbb{N}_k$

under the function f which is the combination function of g and h , is given by $(g(\bar{n}), h(\bar{n}))$

In other words, if the values $g(\bar{n})$ and $h(\bar{n})$ are computable then the additional computational effort required is that for putting these values between a pair of parentheses separated by a comma. However, a machine having these additional capabilities, in addition to the capabilities of the already existing machines for computing $g(\bar{n})$ and $h(\bar{n})$, can easily be constructed. The above informal argument supports the claim that *combination rule* produces computable functions from computable functions.

(ii) **Composition Rule produces computable functions from computable functions.**

Let

$$g: N^k \rightarrow N^m \quad \text{and}$$

$$h: N^m \rightarrow N^p$$

be computable and

$$\bar{x} = (x_1, x_2, \dots, x_k) \in N^k,$$

then g being computable produces through a computational process, some m -tuple say $\bar{y} = (y_1, y_2, \dots, y_m) \in N^m$,

such that $g(\bar{x}) = \bar{y}$

Next, h is computable function with domain N^m and $\bar{y} \in N^m$.

Therefore, the process of getting $h(\bar{y})$ from \bar{y} is computable.

Thus, if we assume computational capabilities already exist for computing $g(\bar{x})$ and $h(\bar{y})$, then for computing the value $h(g(\bar{x}))$ under the composition function of g and h , the only additional computational capability required is that of passing the value $g(\bar{x})$ as an argument to h . This capability can reasonably be assumed. Thus, the computability of the *composition structuring rule* is justified.

Next, we present an argument for the claim that the structuring rule primitive recursion gives computable functions from computable functions.

Let us recall that a function

$$f: N^{k+1} \rightarrow N^m$$

is said to be constructed using primitive recursion from the functions

$$g: N^k \rightarrow N^m \quad \text{and}$$

$$h: N^{k+m+1} \rightarrow N^m,$$

if, for $\bar{x} \in N^k$ and $y \in N$,

$$f(\bar{x}, 0) = g(\bar{x}) \quad \text{and}$$

$$f(\bar{x}, y+1) = h(\bar{x}, y, f(\bar{x}, y)), \quad (32)$$

((32) was used to denote this equation once earlier also).

The claim about computability of f as defined above, is justified by using the Principle of Mathematical Induction on the argument y of $f(\bar{x}, y)$.

Base case. For $y = 0$, as g is computable, therefore, for $\bar{x} \in N^k$, $g(\bar{x})$ and hence $f(\bar{x}, 0)$ is computable.

Induction Hypothesis: Let us assume that for $\bar{x} \in N^k$, and for some $y \in N$, $f(\bar{x}, y)$ is computable.

Induction Step: In view of the assumption under Induction Hypothesis and the fact that h is given to be computable; for $h(\bar{x}, y, f(\bar{x}, y))$ and therefore, for $f(\bar{x}, y+1)$ to be computable, the only additional computational capability required is that of

passing the value of $f(\bar{x}, y)$ as an argument to h . This capability can be reasonably assumed.

Thus, we have *informally* argued in favour of the truth of statement.

Theorem 21: Each primitive recursive function is a total function.

Proof: We know primitive recursive functions are, by definition

- (a) either initial functions
- (b) or the functions obtained by some finite number of applications of the three structuring rules to initial functions.

First, we show initial functions are total:

By definition

- (i) *The Zero function:* $\xi: N^0 \rightarrow N$, is such that $\xi() = 0$
Thus ξ is defined for all elements of its domain N^0 , which is by definition, empty. Thus, ξ is a total function.
- (ii) *the Successor function* $\sigma: N \rightarrow N$ is such that
 $\sigma(x) = x + 1$, for all $x \in N$, the domain.
Thus, successor function is also defined for all elements of its domain N . Thus σ is a total function.

- (iii) *the projection* \prod_i^k with $i, k \in N$, and $i \leq k$
is s.t. if $\bar{x} = (x_1, x_2, \dots, x_1, \dots, x_k) \in N^k$
then $\prod_i^k(\bar{x}) = x_i$ for all $\bar{x} \in N^k$, the domain.

Thus each of the initial functions, is a total function.

Next, we establish that the structuring rules lead from total functions to total functions.

- (i) *The Structuring Rule: Combination*

Let $g: N^k \rightarrow N^m$

$h: N^k \rightarrow N^n$

be two total functions, for which $f: N^k \rightarrow N^{m+n}$ is such that

$$f = g \times h$$

Then by definition of total, for each $\bar{x} = (x_1, x_2, \dots, x_1, \dots, x_k) \in N^k$

$\exists \bar{y} = (y_1, y_2, \dots, y_1, \dots, y_m) \in N^m$ such that

$$g(\bar{x}) = \bar{y} \text{ and}$$

$\exists \bar{z} = (z_1, z_2, \dots, z_1, \dots, z_n) \in N^n$ such that

$$h(\bar{x}) = \bar{z}$$

Thus for each $\bar{x} \in N^k$

$(g(\bar{x}), h(\bar{x})) \in N^{m+n}$

Therefore $f = g \times h$ is total, and hence, combination of two total functions is total.

- (ii) *The Structuring Rule: Composition*

Let functions

$g: N^k \rightarrow N^m$ and

$h: N^m \rightarrow N^n$

be total and $f = h \cdot g$

Then by definition of total, for each $\bar{x} = (x_1, x_2, \dots, x_1, \dots, x_k) \in N^k$

$\exists \bar{y} = (y_1, y_2, \dots, y_i, \dots, y_k) \in N^m$ such that
 $g(\bar{x}) = \bar{y}$ and

further, as h from N^m to N^n is a total function, therefore, for each \bar{y} in N^m , there is a \bar{z} in N^n such that $h(\bar{y}) = \bar{z}$.

But then for each $\bar{x} \in N^k, \exists \bar{z} \in N^n$
such that $(h \cdot g)(\bar{x}) = h(g(\bar{x})) = h(\bar{y}) = \bar{z} \in N^n$

Therefore, $f = h \cdot g: N^k \rightarrow N^n$ is a total function if g and h are total functions.

(iii) *The structuring rule: primitive recursion*

Let $f: N^{k+1} \rightarrow N^m$ be a primitive recursive function which is obtained from the two already defined total functions viz

$$g: N^k \rightarrow N^m \quad \text{and}$$

$$h: N^{k+m+1} \rightarrow N,$$

as follows:

$$f(\bar{x}, 0) = g(\bar{x}) \quad \text{and} \quad (37)$$

$$f(\bar{x}, y+1) = h(\bar{x}, y, f(\bar{x}, y)) \quad \text{for } \bar{x} \in N^k \quad (38)$$

Let $\bar{z} = (x_1, x_2, \dots, x_k, x_{k+1})$ be an arbitrary element of N^{k+1} . We show by induction on the $(k+1)$ th component of \bar{z} that f is total, given that both g and h are total.

Base Case: When $x_{k+1} = 0$

Then from (37), using the fact that g is total we get that f is defined for all $(x_1, x_2, \dots, x_k, 0)$ with $(x_1, \dots, x_k) \in N^k$

Induction Hypothesis: Let us assume that for all $\bar{x} = (x_1, \dots, x_k) \in N^k$ and for $y \in N$, f is defined for (x_1, \dots, x_k, y) .

Induction step: Using the above induction hypothesis and the fact that h is total, the R.H.S of (38) above is defined for all \bar{x} and y . Hence the L.H.S. of (38), i.e., $f(\bar{x}, y+1)$ is total on N^{k+1} .

Hence, primitive recursion leads from total functions to total functions.

Thus, we see that all primitive recursive functions must be total and, as mentioned earlier, computable also.

3.6 PRIMITIVE RECURSION IS WEAK TECHNIQUE

It is natural to ask whether class of all primitive recursive functions cover all computable functions or not? Or in other words, every function, which can be accepted as computable, is also primitive recursive?

The answer to the above is *no*, which is substantiated by the following

Theorem 22: (i) There are computable functions which are not primitive recursive, and even,

(ii) there are total computable functions which are not primitive recursive.

Proof: In order to establish the above, it is sufficient to give an appropriate example for each of the two results.

Example for Theorem part (i) We have established that a primitive recursive function is *necessarily* total. Hence a function which is **not total can not be primitive recursive**.

Consider the following function, which has been discussed earlier.

Quot : $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

s.t.

$$\text{Quot}(x, y) = \begin{cases} z & \text{if } y \neq 0 \text{ and } x = y \cdot z + k \\ & \text{for } 0 \leq k < y \\ \text{undefined} & \text{if } y = 0 \end{cases}$$

is not total, i.e., is strictly partial. Hence Quot can not be *primitive recursive function*.

Example for Theorem part (ii)

The Ackermann's function $A : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ defined below is *total and computable* function but *not primitive recursive*.

$$A(0, y) = y + 1$$

$$A(x + 1, 0) = A(x, 1)$$

$$A(x + 1, y + 1) = A(x, A(x + 1, y))$$

The proof, that A is total and computable but not primitive recursive, is beyond the scope of the course.

Existence Theorems & Their Constructive/Nonconstructive Proofs

Many a theorem is an assertion about the existence of object(s) of a particular type. For example, the assertion, CUBE_SUM: '*There is a positive integer, which can be written as the sum of two cubes of positive integers in two different ways*', if proved true is an example of an existence theorem. There are two ways of proving an existence theorem viz through

- (i) a constructive proof
- (ii) a non-constructive proof.

A **constructive proof** of an existence theorem is *actually about showing* an object of the required type. E.g, writing

$$1729 = 10^3 + 9^3 = 12^3 + 1^3,$$

provides a *constructive proof* of the CUBE-SUM assertion.

In many cases of existence theorems, either it is quite difficult to produce a constructive proof or the constructive proof is not known, then we use a *nonconstructive* method to prove an existence theorem.

Non-Constructive Proof: Sometimes, we do not (rather we are unable to) show the existence of an object of the required type. In such cases, we prove an existence theorem by some non-constructive method of establishing the truth of the existence theorem. A **non-constructive method** shows that *some* element of the required type must exist, but the method is not able to tell exactly which is the element of the required type. We give below two examples of non-constructive proofs of existence theorems.

The first non-constructive existence proof is about the claim: The polynomial equation

$$5x^{1001} + 23x^{93} + 37x^{17} + 52x - 88 = 0$$

has a real root.

The truth of the claim is based on the following well-known result:

A polynomial equation $p(x) = 0$ of degree n and having real coefficients, has n complex roots (not necessarily all distinct) and for each complex root $a + ib$ with $b \neq 0$, $a - ib$ is also a root of $P(x) = 0$.

As a consequence, if $P(x)$ is odd degree, it must have a real root. However, it is quite difficult to find out the real number, which is a real root of the given polynomial equation given above.

The next non-constructive proof is about a well-known result: There exist irrational numbers x and y such that x^y is a rational number.

The following argument establishes the truth of the above result: We know $\sqrt{2}$ is an irrational number, but we do not know whether $(\sqrt{2})^{\sqrt{2}}$ is irrational OR not. If $(\sqrt{2})^{\sqrt{2}}$ is rational then x and y each equal to $\sqrt{2}$ are the required rational numbers. However, if $(\sqrt{2})^{\sqrt{2}}$ is irrational then $x = (\sqrt{2})^{\sqrt{2}}$ and $y = \sqrt{2}$ are two irrational numbers such that $((\sqrt{2})^{\sqrt{2}})^{\sqrt{2}} = (\sqrt{2})^2 = 2$ is rational.

However, in the argument above, we exactly do not know whether the required pair is $(\sqrt{2})^{\sqrt{2}}$ and $\sqrt{2}$ or $\sqrt{2}$ and $\sqrt{2}$.

We give below a non-constructive proof of the theorem: *There is a total computable function which is not primitive recursive.*

Second Proof of Theorem 21 (ii)

All the functions in the following argument are assumed to be of the form $f: \mathbb{N} \rightarrow \mathbb{N}$, only. Let us assume that the above statement is false, i.e., we assume that every total computable function is primitive recursive. Then, we use Cantor's Diagonalization Method, (as is used in showing the existence of a non-rational real number) to arrive at a contradiction.

The representation of a primitive recursive function is obtained by applying finite number of times the structuring rules to the initial functions ξ, σ, \prod^k . The representation of a function which is obtained by an application of a structuring rule to initial functions gives the function as a finite sequence of symbols, e.g., $\sigma \cdot \prod^k (m_1, \dots, m_k)$ uses only finitely many symbols. Each structuring rule adds only finitely many additional symbols to get the representation of a new function from that of already defined function. Thus each primitive recursive function must be representable as a finite sequence of symbols. We arrange the primitive recursive functions according to the number of symbols in the sequence representing the functions, starting with the one with least number of symbols in it, followed by the one having least number of symbols among the remaining. Among function represented by equal number of symbols, we use dictionary type of ordering. Thus, all the sequences of symbols representing the primitive recursive functions can be written in the form of an ordered table starting at the top with a function having least number of symbols in its representation. According to the order of the function in the

table we name the functions, with the top one named as f_1 , next as f_2 and in general n th function in the table being called f_n .

Next, we construct a new function

$$g: \mathbb{N} \rightarrow \mathbb{N} \quad \text{such that} \\ g(n) = f_n(n) + 1 \quad (39)$$

In other words, the value under function g of the argument $n \in \mathbb{N}$, is obtained by taking value under the n th function f_n of n and then adding 1 to it.

As f_n is primitive recursive for each n , the value $f_n(n)$ exists and is obtainable in finite number of steps. Also, adding 1 is only one additional step to get $g(n)$ from $f_n(n)$. Also as f_n is total, therefore for each $n \in \mathbb{N}$, $f_n(n)$ exists and hence $f_n(n)+1$ exists and hence for each $n \in \mathbb{N}$, $g(n)$, being equal to $f_n(n)+1$, exists. Thus $g(n)$ is also total and computable and its value at n differs from the value of f_n because

$$\therefore g(n) = f_n(n) + 1 \neq f_n(n), \\ \text{for each } f_n \text{ in the table.}$$

Thus g is not in the table of all the primitive recursive functions, i.e., g is not primitive recursive.

The last statement contradicts the assumption that every total computable function is primitive. Hence the assumption is wrong, thereby proving the theorem.

Thus, we have proved that the class of primitive recursive functions is a proper subclass of the class of total computable functions.

Thus primitive recursion as a technique for constructing computable functions is weak in the sense that it is not able to construct even such simple functions as Quot. The above discussion suggests that the formal technique of primitive recursion should be further strengthened, so that, the enhanced formal technique captures all such functions which are otherwise, easily seen to be computable. One such technique, called unbounded minimisation, is discussed in the next section.

3.7 THE TECHNIQUES OF UNBOUNDED MINIMALISATION, PARTIAL RECURSION AND μ -RECURSION

In order to achieve the goal mentioned in the previous paragraph, we first consider the

Definition: Unbounded Minimalisation

For a given function

$$g: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$$

we define a function

$$f: \mathbb{N}^k \rightarrow \mathbb{N} \quad \text{such that}$$

for $\bar{x} = (x_1, x_2, \dots, x_k) \in \mathbb{N}^k$ and for some $y \in \mathbb{N}$,

$$f(\bar{x}) \text{ equals } y$$

if (and only if) the following conditions are satisfied.

(i) $g(\bar{x}, y) = 0$ and

(ii) if $g(\bar{x}, z) = 0$ then $y \leq z$

(i.e., y is the smallest among all the values $z \in \mathbb{N}$ for which $g(\bar{x}, z) = 0$)

(iii) $g(\bar{x}, u)$ is defined for all $u \leq y$, with $u \in \mathbb{N}$. (40)

Further, if, for some $\bar{x} \in \mathbb{N}^k$, such a y does not exist, then $f(\bar{x}) = \text{undefined}$. Such a function f is said to be obtained from g through unbounded minimalization and is denoted as

$$f(\bar{x}) = \mu y [g(\bar{x}, y) = 0]$$

Example 22: Let $g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ be defined by the following table.

$g(0, 0) = 5$	$g(1, 0) = 5$	$g(2, 0) = 8$	$g(3, 0) = 1$
$g(0, 1) = 4$	$g(1, 1) = 6$	$g(2, 3) = 0$	$g(3, 1) = 2$
$g(0, 2) = 6$	$g(1, 2) = 0$	$g(2, 1) = 5$	$g(3, 2) = 0$
$g(0, 3) = 0$	$g(1, 3) = 3$	$g(2, 2) = \text{undefined}$	$g(3, 3) = 4$
$g(0, 4) = 1$	$g(1, 4) = 0$	$g(2, 4) = 7$	$g(3, 4) = \text{undefined}$

Then

$$f(0) = 3$$

$f(1) = 2$ (though $g(1, 4) = 0$ also, but 2 is the minimum k such that $f(1, 2) = 0$)

$f(2) =$ is not defined, because $g(2, 3) = 0$, yet $g(2, n)$ is undefined for $n = 2$ which is less than 3.

$f(3) = 2$ (though $g(3, 4)$ is undefined for $y = 4$ but then $4 > 2$ and $g(3, 2) = 0$)

As can be seen from the above example, minimalisation can be defined for functions that are undefined for some values of the domains. Also, minimalisation may produce functions that are undefined for some values of the domain.

Also, unbounded minimalization may lead from total functions $g: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ to partial functions $f: \mathbb{N}^k \rightarrow \mathbb{N}$.

For Example 23:

$$g(n, m) = \begin{cases} m + 1, & \text{for all } m < n \leq 10 \\ 0, & \text{for } m = n \leq 10 \\ n, & \text{otherwise} \end{cases}$$

Then obviously $g: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is total, but, $f: \mathbb{N} \rightarrow \mathbb{N}$ is such that $f(n)$ is not defined for $n \geq 11$.

Also the converse may happen, i.e., unbounded minimalization may lead from some partial functions $g: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ to total functions $f: \mathbb{N}^k \rightarrow \mathbb{N}$. For example

$$\text{Let } g(n, i) = \begin{cases} n - i & \text{for } i \leq n \\ \text{undefined} & \text{if } i > n \end{cases}$$

Then we can see that $f(n) = n$ for all n

Thus unbounded minimalisation leads from a strictly partial function g to a total function f .

Using the technique of unbounded minimalization, we extend the set of computable functions to the class of μ -recursive Functions, also called *Partial Recursive Functions*. The new class includes the class of Primitive Recursive Functions as its proper subclass.

Remarks 24

The reason for the use of the adjective unbounded before minimalization lies in the fact that, there is no bound, on the argument, upto which we are required to try to find a y , which satisfies (i) and (iii) under (40).

Problems with unrestricted application of unbounded minimalisation to a primitive recursive function.

If g is an arbitrary primitive recursive function, then there is no general method of telling whether a y that satisfies all the three conditions of unbounded minimalisation given by (40), exists. In other words, unbounded minimalisation applied to an arbitrary primitive recursive function, *may* not yield a function which may be computable in any intuitive sense (the proof of the claim is beyond the scope of the course).

Remark 26

Bounded Minimalisation: On the lines of the definition of unbounded minimalisation, we can define bounded minimalisation, for a given integer, m , of a partial function

$$g: N^{k+1} \rightarrow N$$

as a function

$$f: N^{k+1} \rightarrow N \quad \text{such that}$$

for $\bar{x} = (x_1, x_2, \dots, x_k) \in N^k$, $m \in N$

and $y \in N$ with $y \leq m$

$f(\bar{x}, m)$ equals y

if (and only if) the following conditions are satisfied:

- (i) $g(\bar{x}, y) = 0$
- (ii) if $g(\bar{x}, z) = 0$ then $y \leq z$
(i.e. y is the smallest among all values $z \in N$ for which $g(\bar{x}, z) = 0$)
- (iii) $g(\bar{x}, u)$ is defined for all $u \leq y$ with $u \in N$

Further, if, for some $\bar{x} \in N^k$, such a $y (\leq m)$, does not exist, then $f(\bar{x}) = \text{undefined}$.

Such a function f is said to be obtained from g through *bounded minimalisation* and is denoted as

$$f(\bar{x}, m) = \mu y \leq m [g(\bar{x}, y)]$$

However, through the following theorem, we show that bounded minimalization is not a powerful technique to extend the class of primitive recursive functions to more general class of computable functions.

Theorem 27: If the function $f(\bar{x}, m)$ is obtained by bounded minimalization for a given integer m and when applied to only primitive recursive function $g(\bar{x}, y)$, then $f(\bar{x}, m)$ must be primitive recursive (however, only the last value may be 'undefined').

Proof: Define the functions

$$h_i(\bar{x}) : N^k \rightarrow N^{i+1}$$

as follows:

$$h_0(\bar{x}) = g(\bar{x}, 0)$$

$$h_1(\bar{x}) = (g(\bar{x}, 0), g_2(\bar{x}, 1))$$

As $h_1(\bar{x})$ is obtained by combination rule applied to values of a primitive recursive function $g(\bar{x}, y)$, therefore, $h_1(\bar{x})$ is a primitive recursive function. Next, consider

$$h_2(\bar{x}) = (h_1(\bar{x}), g(\bar{x}, 2))$$

Again as $h_2(\bar{x})$ is obtained by applying combination rule to two primitive recursive functions $h_1(\bar{x})$ and $g(\bar{x}, 2)$ therefore, $h_2(\bar{x})$ is primitive recursive. Continuing like this, the function $h_1(\bar{x}) \dots h_m(\bar{x})$ are all primitive recursive functions. But

$$h_m(\bar{x}) = (\dots(f(\bar{x}, 0), g(\bar{x}, 1)), g(\bar{x}, 2)) \dots, g(\bar{x}, m))$$

Thus for any \bar{x} , we can compute the row of values $g(\bar{x}, 0), \dots, g(\bar{x}, m)$ can find the minimum $i \leq m$, if it exists, with $g(\bar{x}, i) = 0$

However, if such an i does not exist then also we able to determine that $f(\bar{x}, m)$ is 'undefined'. Thus, we can say that bounded minimalizations of a primitive recursive function is primitive recursive

Remarks 28

At this stage, it is important to note that in unbounded minimalization, the number m is not given and hence, in the case of $f(\bar{x})$ the unbounded minimalization of a given primitive function $g(\bar{x}, y)$, we can not know when to stop finding values $g(\bar{x}, 0), g(\bar{x}, 1) \dots$, if all these values happen to be non-zero, before declaring $f(\bar{x})$ as undefined.

Therefore, we can not claim that the unbounded minimalization of a primitive recursive function is primitive recursive.

Remarks 29

We already know that primitive recursion is a weak computational technique in the sense that it is not able to show even *div* as computable function. Further, through

Theorem 30: we show that *Bounded Minimalisation* produces only primitive recursive functions from primitive recursive functions. Thus *Bounded Minimalisation* can not be used as a technique to extend primitive recursion to more powerful computational technique.

Also, under Remarks 25, we mentioned that *Unbounded Minimalization* though is more powerful technique, yet, its *unrestricted application* may lead to functions which may not be computable in any intuitive sense. Thus we have to find a technique which is a restriction of unbounded minimalisation but is an extension of Bounded Minimalisation. The technique to be described is called μ -recursion or partial recursion. The discussion of partial recursion requires introduction of a number of concepts including the

Definition: Regular Function

A function

$$f: N^{k+1} \rightarrow N$$

is said to be *regular* if (and only if)

for each $\bar{n} \in N^k$, there is an m such that

$$f(\bar{n}, m) = 0$$

In view of the fact that unbounded minimalization may lead from total functions to strictly partial functions, therefore, we need to generalize our definitions of combinations, composition of functions and that of primitive recursion so as to be applicable to strictly partial functions also.

Generalized/New) Combination Rule

Let

$$g: N^k \rightarrow N^m \quad \text{and} \\ h: N^k \rightarrow N^n$$

be two *partial* functions. Then the composition *partial* function

$$f: N^k \rightarrow N^{m+n}$$

is defined as follows:

If $\bar{x} = (x_1, \dots, x_k) \in N^k$

$$f(\bar{x}) = (g(\bar{x}), h(\bar{x})),$$

and both the values $g(\bar{x})$ and then $h(\bar{x})$ are defined; else $f(\bar{x})$ is undefined.

(Generalized/New) Composition Rule

Let $f: N^k \rightarrow N^p$ and

$$g: N^m \rightarrow N^p$$

be *partial* functions then

$$g \cdot f: N^k \rightarrow N^m$$

is given as follows:

Let $\bar{x} = (x_1, x_2, \dots, x_k) \in N^k$

Then $(g \cdot f)(\bar{x}) = (g(f(\bar{x})))$, if both $f(\bar{x})$ and $g(f(\bar{x}))$ are defined, else $g \cdot f$ is undefined.

Similarly, we have the

(Generalized/New) Primitive Recursion:

Given the *partial* functions

$$g: N^k \rightarrow N \text{ and}$$

$$h: N^{k+2} \rightarrow N$$

then a (partial) function

$$f: N^{k+1} \rightarrow N$$

is said to be obtained through partial recursion from g and h , if

$$f(\bar{x}, 0) = g(\bar{x}),$$

including 'undefined' as a possible value for g as well as f and

$$f(\bar{x}, y+1) = h(\bar{x}, y, f(\bar{x}, y)),$$

which will have the value 'undefined' if either $f(\bar{x}, y)$ is 'undefined' or if $f(\bar{x}, y)$ is defined but $h(\bar{x}, y, f(\bar{x}, y))$ is 'undefined'.

Now, we define below the concept of μ -recursion, which as a technique for constructing more complex computable functions, subsumes partial recursion and is more powerful a technique than primitive recursion.

Definition: A μ -recursive function is a partial function (including a total function) that can be constructed from the initial functions by a finite number of applications of the (i) combinations, (ii) compositions, (iii) primitive recursions and (iv) unbounded minimalization to (only) regular functions.

Remarks 31

The fact of primitive recursion technique is a special case of μ -recursion technique, easily follows from the fact that any primitive recursive function f is obtained by finite numbers of applications of (i) combination (ii) composition and (iii) primitive recursion to initial functions. But then by definition of μ -recursion, f must be μ -recursive function (ii) However, μ -recursion is strictly more powerful a technique than primitive recursive from the facts that *div* is not primitive recursive but is μ -recursive as follows from

Example 32: Show the function *quot*: $\mathbb{N}^2 \rightarrow \mathbb{N}$ defined earlier as $\text{div}(x, y) = \{\text{integer portion of } x/y \text{ if } y \neq 0, \text{ undefined if } y = 0\}$, is μ -recursive, but not primitive recursive.

Hint: *quot* is μ -recursive, as $\text{quot}(m, n) = \mu t [(m + 1) \cdot (n + t) = 0]$
Further *div* is a partial function, therefore, it can not be primitive recursive.

Ex. 13) Show that the function *SQRT*: $\mathbb{N} \rightarrow \mathbb{N}$ such that *SQRT*(x) = y if and only if $x = y^2$, is μ -recursive but not primitive recursive.

Finally we come to the end of this unit with **Church's Thesis** which states: The class of μ -recursive functions contains all computable functions. Church's thesis about μ -recursive functions is parallel of Turing thesis about Turing Machines. Church's thesis claims that the μ -recursion technique is ultimate in constructing computable function in the sense that if a function is not μ -recursive then it can not be computable by any formal technique. As mentioned in the previous unit, similar claim is made by Turing Thesis about Turing Machine Model. **We repeat the claim of Turing Thesis:** Turing Machines possess the power of solving any problem that can solved by any computational means. In the next unit, we discuss the equivalence of the two theses giving rise to what is commonly known as Church-Turing Thesis.

3.8 SUMMARY

In this unit, we introduced the *Theory of Recursive Functions*, which is a *declarative* approach to the study of computational phenomenon. We started with some examples of *recursive definitions* of some functions. Then we introduced the concepts of *initial functions* and *primitive recursion* followed by the concept of *primitive recursive function*. An example to motivate the student for the understanding of the concept of primitive recursion, was given before the introduction of the concept of primitive recursion. Next, we exhibited that *primitive recursion is not strong enough a technique* to capture the computational phenomenon, in the sense that some of even elementary functions, though easily seen to be computable, are not primitive-recursive. Then the notion of total computable functions which subsumes the concept of primitive recursive function was introduced, that captures more functions which are intuitively computable. But again it was shown that even the concept of total computable function is not satisfactory in capturing a number of functions which are, intuitively and even formally, computable. Finally, we discussed *μ -recursion* using *unbounded minimalization* technique to capture essentially all the functions which can be shown to be computable by any formal means.

Also, we established the following inclusion relation (\subseteq) between various classes as:
 set of **Initial Functions** \subseteq set of **Primitive Recursive Functions** \subseteq set of **Total computable Functions** \subseteq set of **μ -Recursive Function** \subseteq set of **partial recursive function** \subseteq set of all (partial) functions.

3.9 SOLUTIONS/ANSWERS

Exercise 1 For a function $f: X \rightarrow Y$ to be total, we need to show that for each element x of the domain X , there is an element y of the codomain Y such that $f(x) = y$

ξ is total: $\xi: \mathbb{N} \rightarrow \mathbb{N}$ is such that for each $n \in \mathbb{N}$, the domain there exists $0 \in \mathbb{N}$, the codomain, such that $\xi(n) = 0$. Hence ξ is total

σ is total:

$\sigma: \mathbb{N} \rightarrow \mathbb{N}$ is such that for each $n \in \mathbb{N}$, the domain, there exists $n+1 \in \mathbb{N}$, the codomain, such that $\sigma(n) = n+1$.

Therefore σ is total

$\prod_{1 \leq i \leq k}^k$ is total: By definition

$\prod^k: \mathbb{N}^k \rightarrow \mathbb{N}$

is such that

for an arbitrary element (n_1, n_2, \dots, n_k) of the domain \mathbb{N}^k ,

$\prod_{i=1}^k (n_1, n_2, \dots, n_i, \dots, n_k) = n_i$

Hence \prod^k is total.

Exercise 2 Consider

$$\begin{aligned} &= (\prod_2^2 \times \prod_1^2) \cdot (\prod_1^4 \times \prod_4^4) (8, 7, 4, 2) \\ &= (\prod_2^2 \times \prod_1^2) \cdot (\prod_1^4 (8, 7, 4, 2) \times \prod_4^4 (8, 7, 4, 2)) \\ &= (\prod_2^2 \times \prod_1^2) (8, 2) \\ &= (\prod_2^2 (8, 2) \times \prod_1^2 (8, 2)) \\ &= (2, 8) \end{aligned}$$

Exercise 3 $f(4) = \sigma^3 \cdot f(3)$
 $= \sigma^3 \cdot (\sigma^3 \cdot f(2))$
 $= (\sigma^3 \cdot \sigma^3) \cdot (f(2))$
 $= \sigma^6 \cdot (\sigma^3 \cdot f(1))$
 $= \sigma^9 \cdot (\sigma^3 \cdot f(0))$
 $= \sigma^{12} \cdot f(0) = 12$

Exercise 4 Hint: $n(24)$, take $k=2$, and g as identity function $g: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ i.e., $g(n_1, n_2) = (n_1, n_2)$ for all $n_1, n_2 \in \mathbb{N}$.
 Then (24) takes the form $f(\bar{n}) = g(h_1(\bar{n}), h_2(\bar{n})) = (h_1(\bar{n}), h_2(\bar{n}))$ for all $\bar{n} \in \mathbb{N}^m$,
 Which gives f as combination of h_1 and h_2 .
 Again take $k=1$ and we get f as the Composition of h_1 and g .

Exercise 5 (i) For $m, n \in \mathbb{N}$
 Plus-Prod $(m, n) = (\text{Plus}(m, n), \text{Prod}(m, n))$
 Plus and Prod are already shown to be Primitive recursive. And combination of primitive recursive functions gives a primitive recursive function. Hence the proof

(ii) Exp is primitive recursive follows from the following

$$\text{Exp}(m, 0) = \sigma \cdot \xi(\) \text{ and}$$

$$\text{Exp}(m, n+1) = \text{prod} \left(\prod_1^3 (m, n, \text{Exp}(m, n)), \prod_3^3 (m, n, \text{Exp}(m, n)) \right)$$

(iii) Hint: on the line of Exercise 5 (i)

Exercise 6 $\text{Pred}(0) = \xi(\)$

$$\text{Pred}(1) = \xi$$

$$\text{Pred}(n+1) = \sigma \prod_2^2 (n, \text{pred}(n))$$

Exercise 7 $\text{monus}(m, 0) = m$

$$\text{monus}(m, n+1) = \text{pred}(\text{monus}(m, n))$$

Exercise 8 It can be easily verified that

$$\text{Eq}(m, n) = 1 \stackrel{+}{-} ((m \stackrel{+}{-} n) + (n \stackrel{+}{-} m)),$$

which in formal notation turns out to be

$$\text{Eq}(m, n) = \text{monus}(\sigma \xi(\), \text{plus}(\text{monus}(m, n), \text{monus}(n, m)))$$

$$= (\text{monus} \cdot (\prod_2^2 \times \prod_1^2)) \times \text{monus} \cdot (\prod_1^2 \times \prod_2^2) (m, n)$$

For example

$$\begin{aligned} \text{Eq}(4, 1) &= 1 \stackrel{+}{-} ((4 \stackrel{+}{-} 1) + (1 \stackrel{+}{-} 4)) \\ &= 1 \stackrel{+}{-} (3 + 0) \\ &= 0 \end{aligned}$$

Again

$$\begin{aligned} \text{Eq}(4, 4) &= 1 \stackrel{+}{-} ((4 \stackrel{+}{-} 4) + (4 \stackrel{+}{-} 4)) \\ &= 1 \stackrel{+}{-} (0 + 0) = 1 \end{aligned}$$

Exercise 9 Each integer can be thought of as a member of \mathbb{N}^2 , for example, 5 may be thought of as

(6, 1) and -5 as (1, 6). In general, $(m, n) \in \mathbb{N}^2$ denotes the integer $m - n$.

Then the function *minus*: $I \times I \rightarrow I$ can be sought of as

minus: $(\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N} \times \mathbb{N}$ such that if $(m_1, n_1) \& (m_2, n_2) \in \mathbb{N} \times \mathbb{N}$ then

$$\begin{aligned} \text{minus}((m_1, n_1) \& (m_2, n_2)) \\ &= (m_1 + n_2, n_1 + m_2) \\ &= (\text{plus}(m_1, n_2), \text{plus}(n_1, m_2)) \end{aligned} \tag{39}$$

Plus is already shown to be primitive recursive and combination of two primitive recursive functions (viz plus and plus) is primitive recursive. Hence the above equation (39) shows that *minus* is a primitive recursive function.

Exercise 10 Hint

$$\neg \text{eq} = \text{monus} \cdot (\prod_1^2 \times \text{eq})$$

Exercise 11 Hint

$$K_i = \text{monus}(I_i, I_{i-1})$$

Where

$$I_j(m) = \text{eq}(m \stackrel{+}{-} j, 0)$$

Exercise 12 Hint

$$f = \text{mult}(7, k_0) + \text{mult}(12, k_5) + \text{mult}(8, \text{mult}(\neg K_0, \neg k_5))$$

Exercise 13 As Sqrt is a strictly partial function, therefore, Sqrt is not primitive recursive.

Further as $\text{Sqrt}(x) = \mu t[(1-t) \text{Eq}(x, \text{prod}(t,t))=0]$, therefore, Sqrt is μ -recursive.

3.10 FURTHER READINGS

Lewis H.R and Papadimitriou C.H., *Elements of the Theory of Computation* PHI (1981),

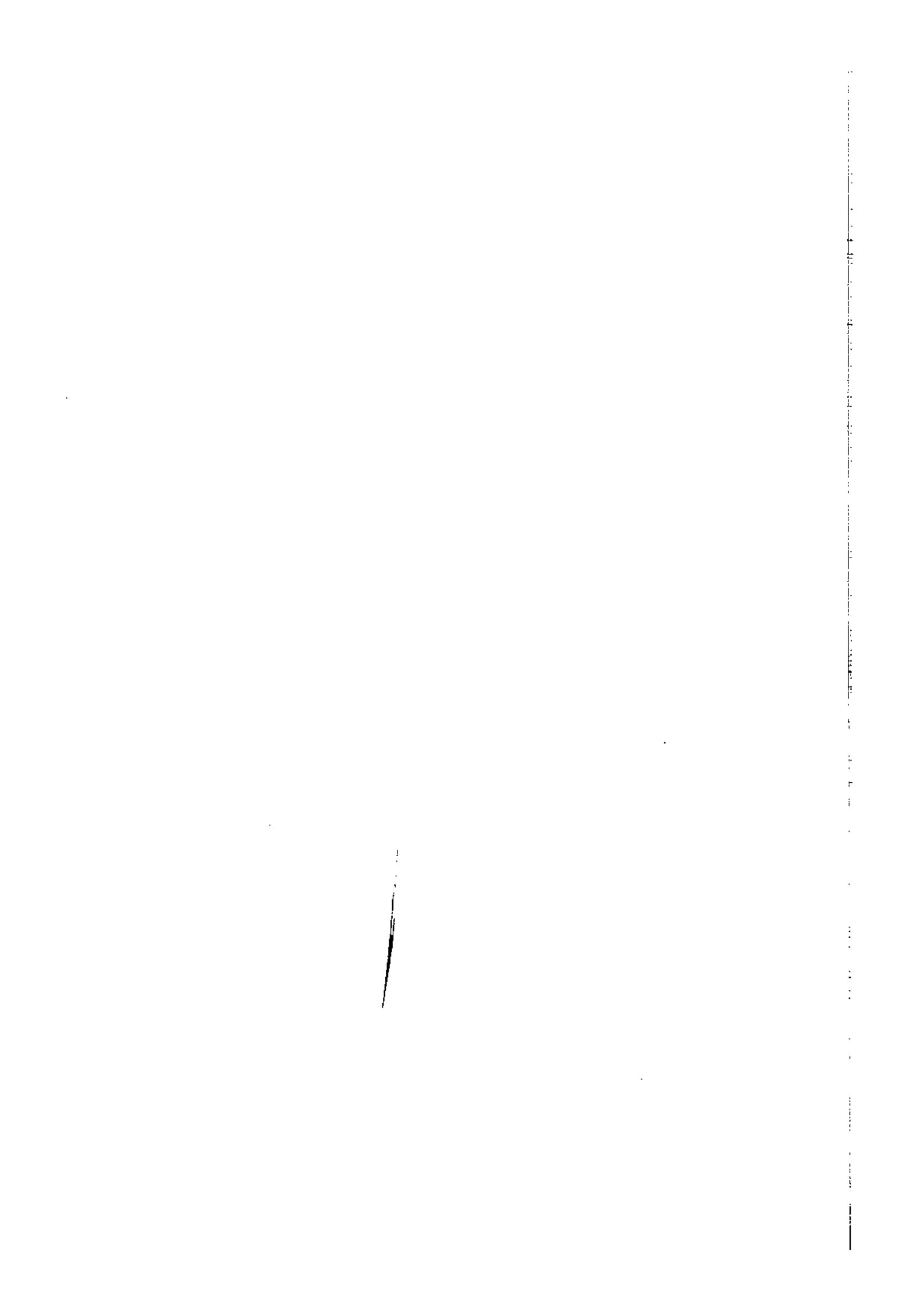
Peter R., *Recursive Functions*, Academic Press (1967)

Epstein Richard L. and Carmill W.A., *Computability, Computable Functions, Logic and the Foundations of Mathematics* (11 Edition) Wadsworth & Brooks (2000).

(A highly readable book presenting advanced level topics from elementary point of view)

Goodstein R.L., *Recursive Analysis* North-Holland (1961)

Rogers H., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill (1967)





Uttar Pradesh
Rajarshi Tandon Open University

BCA-18
THEORY OF
COMPUTATION

Block

3

COMPLEXITY AND COMPUTABILITY

UNIT 1

Computability/Decidability **5**

UNIT 2

Complexity **19**

UNIT 3

Applications **47**

BLOCK INTRODUCTION

"Both knowledge and wisdom extend man's reach. Knowledge led to computers, wisdom to chopsticks. Unfortunately our association is overinvolved with the former. The latter will have to wait for a more sublime day".

Alan J. Perlis in his 1966 Turing Award Lecture
(The Turing Award was given for the first time in 1966.)

Let us wait for some more time for chowmein and chopsticks and, instead, have some other nugget of wisdom:

"Kurt Gödel's incompleteness theorems tell us that rational thought can never penetrate to the final, ultimate truth... Gödel was one of the undisputed mathematical geniuses of the twentieth century and his theorems are pure mathematics, the ultimate precision in reason..."

B.K. Banerjee
in

'Primacy of Faith over Reason'
Times of India May 10, 1993.

After having agreed to repose our *faith* in Turing-Church thesis, according to which Turing Machine and, hence, all its equivalents from other approaches, are ultimate models of computation; let us investigate what (type of) problems are computable, i.e., (computationally) solvable. Further, out of the solvable problems, we study the issue of relative degrees of difficulty in solving the solvable problems.

In Unit 1, we discuss various issues related to the problems, which are not solvable by any computational means. Such problems are also called undecidable problems.

Out of the problems that are solvable by some computational means, there are problems which are not feasible in the sense that such problems require very large amount of computational resources. **In Unit 2**, we classify computationally solvable problems into various categories like P, NP, intractable, NP-Hard, and NP-complete problems. Also, we discuss a technique for establishing some of the well known problems etc as NP-complete.

Finally, **in Unit 3**, We discuss some of the applications of the various topics covered in all the three blocks of this course.

.....

UNIT 1 COMPUTABILITY/DECIDABILITY

Structure	Page Nos.
0 Introduction	5
1 Objectives	6
2 Decidable and Undecidable problems	6
3 The Halting Problem	7
4 Reduction to Another Undecidable Problem	10
5 Undecidability of Post Correspondence Problem	12
6 Undecidable Problems for Context Free languages	13
7 Other Undecidable Problems	14
8 Summary	16
9 Solutions/Answers	16
10 Further Readings	18

0 INTRODUCTION

Most of the time, the computers are talked in respect of their wonderful achievements and, of course, once in a while, also about the blunders committed by some computer system, e.g., of withdrawing or depositing more than 99 million dollars from a bank account against the required 99 dollars only). **However, most of the non-specialists are not aware of the general limitations of computers.** One very important fact, in this respect, is that there are large number of problems which no computer, including any one that may be designed and developed at any time in the future, is and will ever be able to solve. *Rather, the number of problems that can be solved computationally is much less than the number of problems that can never be solved using only computational means.* In this unit, we discuss issues and problems that exhibit the limitations of computing devices in solving problems.

In this sense, we explore the limits on the capabilities of computers. *We also prove one of the deepest results in computer science: the undecidability of the halting problem.* Alan Turing first proved this result in 1936. It is related to Gödel's incompleteness Theorem which states that there is no system of logic strong enough to prove all true sentences of number theory. Essentially, Gödel uses a fixpoint construction to construct a self-referential sentence of number theory which states something to the effect: *"I am not provable"*. The argument is quite complex. However, the argument is basically analogous to the one given in support of the fact that the truth value of the statement *'I am telling lies'* can not be determined.

In view of the large number of applications of modern computer systems that help us in solving problems from almost every domain of human experience, you might be tempted to think that computers can solve any problem if the problem is properly formulated. You'd soon find that there are problems, even from a highly formal discipline like mathematics, which can be properly formulated, but can not be solved by computational means to through computational means those problems from disciplines like social science's, philosophy or religion etc. that can't even be formulated as computational problems.

We will discuss problems, which though can be formulated properly, yet are not solvable through any computational means. And we will *prove* that such problems cannot be solved no matter

- what language is used?
- what machine is used?

...as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now that we have gigantic computers, programming has become an equally gigantic problem. *In this sense the electronic industry has not solved a single problem; it has only created them* – it has created the problem of using its products. To put in another way: ... society's ambition to apply these machines grew in proportion and it is the programmer ...

Edsger W. Dijkstra
In
Turing Award Lecture
(1972)

- much computational resources are devoted in attempting to solve the problem etc.

For problems that can not be solved by computational means, we can *approximate* their solutions, but it's impossible to get the perfectly correct solutions in all cases.

One of the important problem among all such problems is the **halting problem**: given a program and an input, does the program halt when applied to that input?

Answer: it's impossible to determine in general. However, there may be some special cases for which you may get the answer, but there is no general algorithm that works in all cases, and provably so.

1.1 OBJECTIVES

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

- show that Halting Problem is uncomputable/unsolvable/undecidable;
- to explain the general technique of *Reduction* to establish other problems as uncomputable;
- establish unsolvability of many unsolvable problems using the technique of reduction;
- enumerate large number of unsolvable problems, including those about Turing Machines and about various types of grammars/languages including context-free, context-sensitive and unrestricted etc.

1.2 DECIDABLE AND UNDECIDABLE PROBLEMS

A function g with domain D is said to be computable if there exists some Turing machine

$M = (Q, \Sigma, T, \delta, q_0, F)$ such that
 $q_0 w \vdash^* q_f g(w), \quad q_f \in F, \text{ for all } w \in D.$

where

$q_0 \omega$ denotes the initial configuration with left-most symbol of the string ω being scanned in state q_0 and $q_f g(\omega)$ denotes the final c.

A function is said to be uncomputable if no such machine exists. There may be a Turing machine that can compute f on part of its domain, but we call the function computable only if there is a Turing machine that computes the function on the whole of its domain.

For some problems, we are interested in simpler solution in terms of "yes" or "no". For example, we consider problem of context free grammar i.e., for a context free grammar G , Is the language $L(G)$ ambiguous. For some G the answer will be "yes", for others it will be "no", but clearly we must have one or the other. The problem is to decide whether the statement is true for any G we are given. The domain for this problem is the set of all context free grammars. We say that a problem is decidable if there exists a Turing machine that gives the correct answer for every statement in the domain of the problem.

Similarly, consider the problem of equivalence of context free grammar i.e., to determine whether two context free grammars are equivalent. Again, given context free grammars G_1 and G_2 , the answer may be "yes" or "no". The problem is to

decide whether the statement is true for any two given context free grammars G_1 and G_2 . The domain for this problem is the set of all context free grammars. We say that a problem is decidable if there exists a Turing machine that gives the correct answer for every statement in the domain of the problem.

A class of problems with two output: "yes" or "no" is said to be decidable (solvable) if there exists some definite algorithm which always terminates (halts) with one of two outputs "yes" or "no". Otherwise, the class of problems is said to be undecidable (unsolvable).

1.3 THE HALTING PROBLEM

There are many problem which are not computable. But, we start with a problem which is important and that at the same time gives us a platform for developing later results. One such problem is the halting problem. Algorithms may contain loops that may be infinite or finite in length. The amount of work done in an algorithm usually depends on the data input. Algorithms may consist of various numbers of loops, nested or in sequence. Informally, the Halting problem can be put as:

Given a Turing machine M and an input w to the machine M , determine if the machine M will eventually halt when it is given input w .

Trial solution: Just run the machine M with the given input w .

- If the machine M halts, we know the machine halts.
- But if the machine doesn't halt in a reasonable amount of time, we cannot conclude that it won't halt. Maybe we didn't wait long enough.

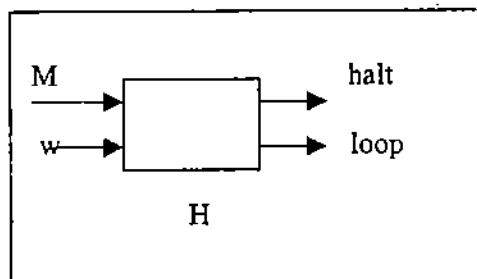
What we need is an algorithm that can determine the correct answer for any M and w by performing some analysis on the machine's description and the input. But, we will show that no such algorithm exists.

Let us see first, proof devised by Alan Turing (1936) that halting problem is unsolvable.

Suppose you have a solution to the halting problem in terms of a machine, say, H . H takes two inputs:

1. a program M and
2. an input w for the program M .

H generates an output "halt" if H determines that M stops on input w or it outputs "loop" otherwise.



So now H can be revised to take M as both inputs (the program and its input) and H should be able to determine if M will halt on M as its input.

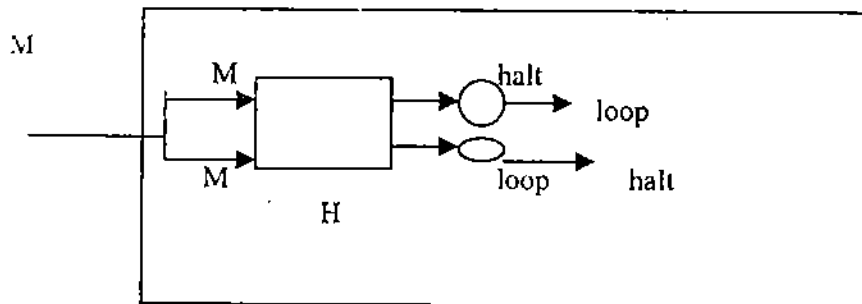
...the programming task is (still) an intellectual challenge of the highest caliber...How not to get lost in the complexities of our own making is still computing's core challenge...

Edger W. Dijkstra
in
Postscript (1986) to
his 1972 Turing
Award Lecture

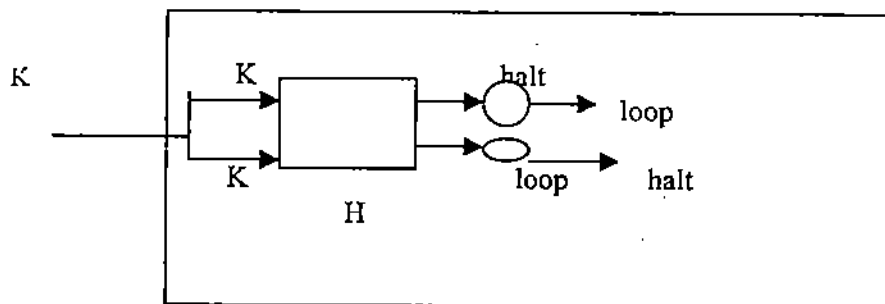
Let us construct a new, simple algorithm K that takes H's output as its input and does the following:

1. if H outputs "loop" then K halts,
2. otherwise H's output of "halt" causes K to loop forever.

That is, K will do the opposite of H's output.



Since K is a program, let us use K as the input to K.



If H says that K halts then K itself would loop (that's how we constructed it).
If H says that K loops then K will halt.

In either case H gives the wrong answer for K. Thus H cannot work in all cases.

We've shown that it is possible to construct an input that causes any solution H to fail. Hence, The halting problem is undecidable.

Now, we formally define what we mean by the halting problem.

Definition 1.1: Let W_M be a string that describes a Turing machine $M = (Q, \Sigma, T, \delta, q_0, F)$, and let w be a string in Σ^* . We will assume that W_M and w are encoded as a string of 0's and 1's. A solution of the halting problem is a Turing machine H, which for any W_M and w , performs the computation

$$q_0 W_M w \vdash^* x_1 q_y x_2 \text{ if } M \text{ applied to } w \text{ halts, and}$$

$$q_0 W_M w \vdash^* y_1 q_n y_2 \text{ if } M \text{ applied to } w \text{ does not halt.}$$

Here q_y and q_n are both final states of H.

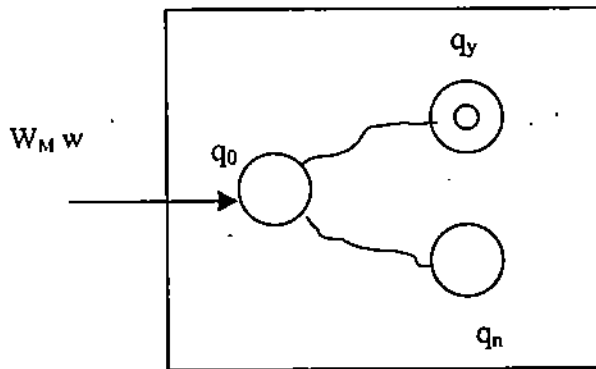
Theorem 1.1: There does not exist any Turing machine H that behaves as required by Definition 1.1. The halting problem is therefore undecidable.

Proof: We provide proof by contradiction. Let us assume that there exists an algorithm, and consequently some Turing machine H, that solves the halting problem. The input to H will be the string $W_M w$. The requirement is then that, the Turing machine H will halt with either a yes or no answer. We capture this by asking

that H will halt in one of two corresponding final states, say, q_y or q_n . As per Definition 8.1, we want H to operate according to the following rules:

$$\begin{aligned} q_0 W_M w & \xrightarrow{-}_M x_1 q_y x_2 && \text{if } M \text{ applied to } w \text{ halts, and} \\ q_0 W_M w & \xrightarrow{-}_M y_1 q_n y_2 && \text{if } M \text{ applied to } w \text{ does not halt.} \end{aligned}$$

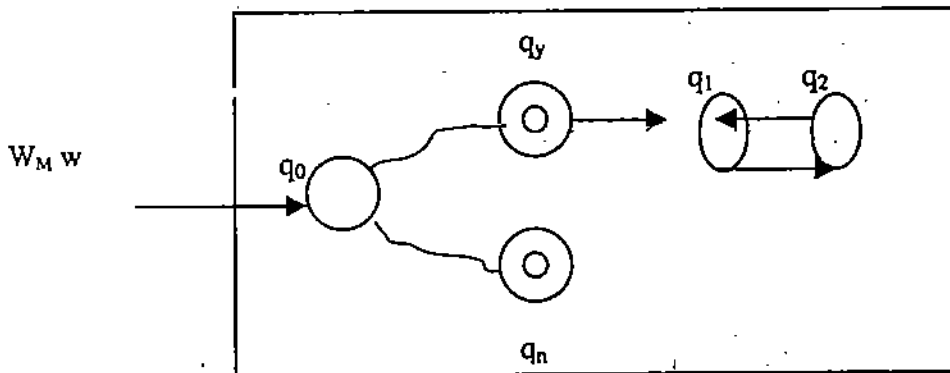
This situation can also be visualized by a block diagram given below:



Next, we modify H to produce H_1 such that

- If H says that it will halt then H_1 itself would loop
- If H says that H will not halt then H_1 will halt.

We can achieve this by adding two more states say, q_1 and q_2 . Transitions are defined from q_y to q_1 , from q_1 to q_2 and from q_2 to q_1 , regardless of the tape symbol, in such a way that the tape remains unchanged. This is shown by another block diagram given below:



Formally, the action of H_1 is described by

$$\begin{aligned} q_0 W_M w & \xrightarrow{-}_{H_1} \infty && \text{if } M \text{ applied to } w \text{ halts, and} \\ q_0 W_M w & \xrightarrow{-}_{H_1} y_1 q_n y_2 && \text{if } M \text{ applied to } w \text{ does not halt.} \end{aligned}$$

Here, ∞ stands for Turing machine is in infinite loop i.e., Turing machine will run forever. Next, we construct another Turing machine H_2 from H_1 . This new machine takes as input W_M and copies it, ending in its initial state q_0 . After that, it behaves exactly like H_1 . The action of H_2 is such that

$$\begin{aligned} q_0 W_M & \xrightarrow{-}_{H_2} q_0 W_M W_M && \xrightarrow{-}_{H_2} \infty && \text{if } M \text{ applied to } W_M \text{ halts, and} \\ q_0 W_M & \xrightarrow{-}_{H_2} y_1 q_n y_2 && \text{if } H_2 \text{ applied to } W_M \text{ does not halt.} \end{aligned}$$

This clearly contradicts what we assumed. In either case H_2 gives the wrong answer for W_M . Thus H cannot work in all cases.

We've shown that it is possible to construct an input that causes any solution H to fail. Hence, the halting problem is undecidable.

Theorem 2.2: If the halting problem were decidable, then every recursively enumerable language would be recursive. Consequently, the halting problem is undecidable.

Proof: Recall that

1. A language is *recursively enumerable* if there exists a Turing machine that accepts every string in the language and does not accept any string not in the language.
2. A language is *recursive* if there exists a Turing machine that accepts every string in the language and rejects every string not in the language.

Let L be a recursively enumerable language on Σ , and let M be a Turing machine that accepts L . Let us assume H be the Turing machine that solves the halting problem. We construct from this the following algorithm:

1. Apply H to $W_M w$. If H says "no", then by definition w is not in L .
2. If H says "yes", then apply M to w . But M must halt, so it will ultimately tell us whether w is in L or not.

This constitutes a membership algorithm, making L recursive. But, we know that there are recursively enumerable languages that are not recursive. The contradiction implies that H cannot exist *i.e.*, the halting problem is undecidable.

1.4 REDUCTION TO ANOTHER UNDECIDABLE PROBLEM

Once we have shown that the halting problem is undecidable, we can show that a large class of other problems about the input/output behaviour of programs are undecidable.

Examples of undecidable problems

- **About Turing machines:**
 - Is the language accepted by a TM empty, finite, regular, or context-free?
 - Does a TM meet its "specification ?," that is, does it have any "bugs."
- **About Context Free languages**
 - Are two context-free grammars equivalent?
 - Is a context-free grammar ambiguous?

Not so surprising, Although this result is sweeping in scope, maybe it is not surprising. If a simple question such as whether a program halts or not is undecidable, why should one expect that any other property of the input/output behavior of programs is decidable? Rice's theorem makes it clear that failure to decide halting implies failure to decide any other interesting question about the

input/output behaviour of programs. Before we consider Rice's theorem, we need to understand the concept of problem reduction on which its proof is based.

Reducing problem B to problem A means finding a way to convert problem B to problem A, so that a solution to problem A can be used to solve problem B.

One may ask, Why is this important? A reduction of problem B to problem A shows that problem A is at least as difficult to solve as problem B. Also, we can show the following:

- To show that a problem A is undecidable, we reduce another problem that is known to be undecidable to A.
- Having proved that the halting problem is undecidable, we use problem reduction to show that other problems are undecidable.

Example 1: Totality Problem

Decide whether an arbitrary TM halts on all inputs. (If it does, it computes a "total function"). This is equivalent to the problem of whether a program can ever enter an infinite loop, for any input. It differs from the halting problem, which asks whether it enters an infinite loop for a particular input.

Proof: We prove that the halting problem is reducible to the totality problem. That is, if an algorithm can solve the totality problem, it can be used to solve the halting problem. Since no algorithm can solve the halting problem, the totality problem must also be undecidable.

The reduction is as follows. For any TM M and input w , we create another TM M_1 that takes an arbitrary input, ignores it, and runs M on w . Note that M_1 halts on all inputs if and only if M halts on input w . Therefore, an algorithm that tells us whether M_1 halts on all inputs also tells us whether M halts on input w , which would be a solution to the halting problem.

Hence, The totality problem is undecidable.

Example 2: Equivalence problem

Decide whether two TMs accept the same language. This is equivalent to the problem of whether two programs compute the same output for every input.

Proof: We prove that the totality problem is reducible to the equivalence problem. That is, if an algorithm can solve the equivalence problem, it can be used to solve the totality problem. Since no algorithm can solve the totality problem, the equivalence problem must also be unsolvable.

The reduction is as follows. For any TM M , we can construct a TM M_1 that takes any input w , runs M on that input, and outputs "yes" if M halts on w . We can also construct a TM M_2 that takes any input and simply outputs "yes." If an algorithm can tell us whether M_1 and M_2 are equivalent, it can also tell us whether M_1 halts on all inputs, which would be a solution to the totality problem.

Hence, the equivalence problem is undecidable.

Practical implications

- The fact that the totality problem is undecidable means that we cannot write a program that can find any infinite loop in any program.

- The fact that the equivalence problem is undecidable means that the code optimization phase of a compiler may improve a program, but can never guarantee finding the optimally efficient version of the program. There may be potentially improved versions of the program that it cannot even be sure are equivalent.

We now describe a more general way of showing that a problem is undecidable i.e., **Rice's theorem**. First we introduce some definitions.

- A *property* of a program (TM) can be viewed as the set of programs that have that property.
- A *functional (or non-trivial) property* of a program (TM) is one that some programs have and some don't.

Rice's theorem (proof is not required)

- "Any functional property of programs is undecidable."
- A functional property is:
 - (i) a property of the input/output behaviour of the program, that is, it describes the mathematical function the program computes.
 - (ii) nontrivial, in the sense that it is a property of some programs but not all programs.

Examples of functional properties

- The language accepted by a TM contains at least two strings.
- The language accepted by a TM is empty (contains no strings).
- The language accepted by a TM contains two different strings of the same length.

Rice's theorem can be used to show that whether the language accepted by a Turing machine is context-free, regular, or even finite, are undecidable problems. Not all properties of programs are functional.

1.5 UNDECIDABILITY OF POST CORRESPONDENCE PROBLEM

Undecidable problems arise in language theory also. It is required to develop techniques for proving particular problems undecidable. In 1946, Emil Post proved that the following problem is undecidable:

Let Σ be an alphabet, and let L and M be two lists of nonempty strings over Σ , such that L and M have the same number of strings. We can represent L and M as follows:

$$L = (w_1, w_2, w_3, \dots, w_k)$$
$$M = (v_1, v_2, v_3, \dots, v_k)$$

Does there exist a sequence of one or more integers, which we represent as (i, j, k, \dots, m) , that meet the following requirements:

- Each of the integers is greater than or equal to one.

- Each of the integers is less than or equal to k . (Recall that each list has k strings).
- The concatenation of w_i, w_j, \dots, w_m is equal to the concatenation of v_i, v_j, \dots, v_m .

If there exists the sequence (i, j, k, \dots, m) satisfying above conditions then (i, j, k, \dots, m) is a solution of PCP.

Let us consider some examples.

Example 3: Consider the following instance of the PCP:

Alphabet $\Sigma = \{ a, b \}$
List $L = (a, ab)$
List $M = (aa, b)$

We see that $(1, 2)$ is a sequence of integers that solves this PCP instance, since the concatenation of a and ab is equal to the concatenation of aa and b (i.e. $w_1 w_2 = v_1 v_2 = aab$). other solutions include: $(1, 2, 1, 2), (1, 2, 1, 2, 1, 2)$ and so on.

Example 4: Consider the following instance of the PCP Alphabet $\Sigma = \{ 0, 1 \}$

List $L = (0, 01000, 01)$
List $M = (000, 01, 1)$

A sequence of integers that solves this problem is $(2, 1, 1, 3)$, since the concatenation of $01000, 0, 0$ and 01 is equal to the concatenation of $01, 000, 000$ and 1 (i.e., $w_2 w_1 w_1 w_3 = v_2 v_1 v_1 v_3 = 010000001$).

1.6 UNDECIDABLE PROBLEMS FOR CONTEXT-FREE LANGUAGES

The Post correspondence problem is a convenient tool to study undecidable questions for context free languages. We illustrate this with an example.

Theorem 1.2: There exists no algorithm for deciding whether any given context-free grammar is ambiguous.

Proof: Consider two sequences of strings $A = (u_1, u_2, \dots, u_m)$ and $B = (v_1, v_2, \dots, v_m)$ over some alphabet Σ . Choose a new set of distinct symbols a_1, a_2, \dots, a_m such that

$$\{a_1, a_2, \dots, a_m\} \cap \Sigma = \emptyset,$$

and consider the two languages

$$L_A = \{ u_i u_j \dots u_i u_k a_k a_1 \dots, a_j a_i \} \text{ defined over } A \text{ and } \{a_1, a_2, \dots, a_m\}$$

and

$$L_B = \{ v_i v_j \dots v_i v_k a_k a_1 \dots, a_j a_i \} \text{ defined over } B \text{ and } \{a_1, a_2, \dots, a_m\}.$$

Let G be the context free grammar given by

$$(\{S, S_A, S_B\}, \{a_1, a_2, \dots, a_m\} \cup \Sigma, P, S)$$

where the set of productions P is the union of the two subsets: the first set P_A consists of

$$\begin{aligned} S &\rightarrow S_A, \\ S_A &\rightarrow u_i S_A a_i \mid u_i a_i, \quad i = 1, 2, \dots, n, \end{aligned}$$

the second set P_B consists of

$$\begin{aligned} S &\rightarrow S_B, \\ S_B &\rightarrow v_i S_B a_i \mid v_i a_i, \quad i = 1, 2, \dots, n. \end{aligned}$$

Now take

$$G_A = (\{S, S_A\}, \{a_1, a_2, \dots, a_n\} \cup \Sigma, P_A, S)$$

and

$$G_B = (\{S, S_B\}, \{a_1, a_2, \dots, a_n\} \cup \Sigma, P_B, S)$$

Then,

$$\begin{aligned} L_A &= L(G_A), \\ L_B &= L(G_B), \end{aligned}$$

and

$$L(G) = L_A \cup L_B.$$

It is easy to see that G_A and G_B by themselves are unambiguous. If a given string in $L(G)$ ends with a_i , then its derivation with grammar G_A must have started with $S \Rightarrow u_i S_A a_i$. Similarly, we can tell at any later stage which rule has to be applied. Thus, if G is ambiguous it must be because there is w for which there are two derivations

$$S \Rightarrow S_A \Rightarrow u_i S_A a_i \Rightarrow^* u_i u_j \dots u_k a_k \dots a_j a_i = w$$

and

$$S \Rightarrow S_B \Rightarrow v_i S_B a_i \Rightarrow^* v_i v_j \dots v_k a_k \dots a_j a_i = w.$$

Consequently, if G is ambiguous, then the Post correspondence problem with the pair (A, B) has a solution. Conversely, if G is unambiguous, then the Post correspondence problem cannot have solution.

If there existed an algorithm for solving the ambiguity problem, we could adapt it to solve the Post correspondence problem. But, since there is no algorithm for the Post correspondence problem, we conclude that the ambiguity problem is undecidable.

1.7 OTHER UNDECIDABLE PROBLEMS

- Does a given Turing machine M halt on all inputs?
- Does Turing machine M halt for *any* input? (That is, is $L(M) = \emptyset$?)
- Do two Turing machines M_1 and M_2 accept the same language?
- Is the language $L(M)$ finite?
- Does $L(M)$ contain any two strings of the same length?
- Does $L(M)$ contain a string of length k , for some given k ?

- If G is a unrestricted grammar.
- Does $L(G) = \emptyset$?
- Does $L(G)$ infinite ?
- If G is a context sensitive grammar.
- Does $L(G) = \emptyset$?
- Does $L(G)$ infinite ?
- If L_1 and L_2 are any context free languages over Σ .
- Does $L_1 \cap L_2 = \emptyset$?
- Does $L_1 = L_2$?
- Does $L_1 \subseteq L_2$?
- If L is recursively enumerable language over Σ .
- Does L empty ?
- Does L finite ?

Ex. 1) Show that the state-entry problem is undecidable.

Hint: The problem is described as follows: Given any Turing machine $M = (Q, \Sigma, T, \delta, q_0, F)$ and any $q \in Q, w \in \Sigma^+$, to determine whether Turing machine M , when given input w , ever enters state q .

Ex. 2) Show that the blank tape halting problem is undecidable.

Hint: The problem is described as follows: Given a Turing machine M , Does Turing machine M halts when given a blank input tape?

Ex. 3) Consider the following instance of the PCP:

Alphabet $\Sigma = \{ 0, 1, 2 \}$

List $L = (0, 1, 2)$

List $M = (00, 11, 22)$

Does PCP have a solution ?

Ex. 4) Consider the following instance of the PCP:

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

List $L = (ba, abb, bab)$

List $M = (bab, bb, abb)$

Does PCP have a solution ?

Ex. 5) Does PCP with two lists $A = (b, babbb, ba)$ and $B = (bbb, ba, a)$ have a solution ?

Ex. 6) Does PCP with two lists $A = (ab, b, b)$ and (abb, ba, bb) have a solution ?

Ex.7) Show that there does not exist algorithm for deciding whether or not

$L(G_A) \cap L(G_B) = \emptyset$ for arbitrary context free grammars G_A and G_B .

1.8 SUMMARY

- A decision problem is a problem that requires a yes or no answer. A decision problem that admits no algorithmic solution is said to be undecidable.
- No undecidable problem can ever be solved by a computer or computer program of any kind. In particular, there is no Turing machine to solve an undecidable problem.
- We have not said that undecidable means we don't know of a solution today but might find one tomorrow. It means we can never find an algorithm for the problem.
- We can show no solution can exist for a problem A if we can reduce it into another problem B and problem B is undecidable.

1.9 SOLUTIONS/ANSWERS

Exercise 1

The problem is described as follows: Given any Turing machine $M = (Q, \Sigma, T, \delta, q_0, F)$ and any $q \in Q, w \in \Sigma^+$, to determine whether Turing machine M , when given input w , ever enters state q .

The problem is to determine whether Turing machine M , when given input w , ever enters state q .

The only way a Turing machine M halts is if it enters a state q for which some transition function $\delta(q_i, a_i)$ is undefined. Add a new final state Z to the Turing machine, and add all these missing transitions to lead to state Z .

Now use the (assumed) state-entry procedure to test whether state Z is ever entered when M is given input w . This will reveal whether the original machine M halts. We conclude that it must not be possible to build the assumed state-entry procedure.

Exercise 2

It is another problem which is undecidable. The problem is described as follows: Given a Turing machine M , does Turing machine M halt when given a blank input tape?

Here, we will reduce the blank tape halting problem to the halting problem. Given M and w , we first construct from M a new machine M_w that starts with a blank tape, writes w on it, then positions itself in configuration q_0w . After that, M_w acts exactly like M . Hence, M_w will halt on a blank tape if and only if M halts on w .

Suppose that the blank tape halting problem were decidable. Given any M and w , we first construct M_w , then apply the blank tape halting problem algorithm to it. The conclusion tells us whether M applied to w will halt. Since this can be done for any M and w , an algorithm for the blank tape halting problem can be converted into an algorithm for the halting problem. Since the halting problem is undecidable, the same must be true for the blank tape halting problem.

Exercise 3

There is no solution to this problem, since, any potential solution, the

concatenation of the strings from list L will contain half as many letters as the concatenation of the corresponding strings from list M.

Exercise 4

We can not have string beginning with $w_2 = abb$ as the counterpart $v_2 = bb$ exists in another sequence and first character does not match. Similarly, no string can begin with $w_3 = bab$ as the counterpart $v_3 = abb$ exists in another sequence and first character does not match. The next choice left with us is start the string with $w_1 = ba$ from L and the counterpart $v_1 = bab$ from M. So, we have

ba

bab

The next choice from L must begin with b. Thus, either we choose w_1 or w_3 as their string starts with symbol b. But, the choice of w_1 will make two string look like:

baba

babbab

While the choice of w_3 direct to make choice of v_3 and the string will look like:

babab

bababb

Since the string from list M again exceeds the string from list L by the single symbol b, a similar argument shows that we should pick up w_3 from list L and v_3 from list M. Thus, there is only one sequence of choices that generates compatible strings, and for this sequence string M is always one character longer. Thus, this instance of PCP has no solution.

Exercise 5

We see that (2, 1, 1, 3) is a sequence of integers that solves this PCP instance, since the concatenation of babb, b, b and ba is equal to the concatenation of ba, bbb, bbb and a (i.e., $w_2 w_1 w_1 w_3 = v_2 v_1 v_1 v_3 = \text{babbbbbba}$).

Exercise 6

For each string in A and corresponding string in B, the length of string of A is less than counterpart string of B for the same sequence number. Hence, the string generated by a sequence of strings from A is shorter than the string generated by the sequence of corresponding strings of B. Therefore, the PCP has no solution.

Exercise 7

Proof: Consider two grammars

$$G_A = (\{S_A\}, \{a_1, a_2, \dots, a_m\} \cup \Sigma, P_A, S_A)$$

and

$$G_B = (\{S_B\}, \{a_1, a_2, \dots, a_m\} \cup \Sigma, P_B, S_B).$$

where the set of productions P_A consists of

$$S_A \rightarrow u_i S_A a_i \mid u_i a_i, \quad i = 1, 2, \dots, n,$$

and the set of productions P_0 consists of

$$S_B \rightarrow v_i S_B a_i \mid v_i a_i, \quad i = 1, 2, \dots, n.$$

where consider two sequences of strings $A = (u_1, u_2, \dots, u_m)$ and $B = (v_1, v_2, \dots, v_m)$ over some alphabet Σ . Choose a new set of distinct symbols a_1, a_2, \dots, a_m , such that

$$\{a_1, a_2, \dots, a_m\} \cap \Sigma = \emptyset,$$

Suppose that $L(G_A)$ and $L(G_B)$ have a common element, i.e.,

$$S_A \Rightarrow u_i S_A a_i \Rightarrow^* u_i u_j \dots u_k a_k \dots a_j a_i$$

and

$$S_B \Rightarrow v_i S_B a_i \Rightarrow^* v_i v_j \dots v_k a_k \dots a_j a_i.$$

Then the pair (A, B) has a PC-solution. Conversely, if the pair does not have a PC-solution, then $L(G_A)$ and $L(G_B)$ cannot have a common element. We conclude that $L(G_A) \cap L(G_B)$ is nonempty if and only if (A, B) has a PC-solution.

1.10 FURTHER READINGS

1. H.R. Lewis & C.H. Papadimitriou: *Elements of the Theory of computation*, PHI, (1981).
2. J.E. Hopcroft, R. Motwani & J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation* (II Ed.) Pearson Education Asia (2001).
3. J.E. Hopcroft and J.D. Ullman: *Introduction to Automata Theory, Language, and Computation*, Narosa Publishing House (1987).
4. J.C. Martin: *Introduction to Languages and Theory of Computation*, Tata-Mc Graw-Hill (1997).
5. M.R. Garey & D.S. Johnson: *Computers and Intractability – A Guide to the theory of NP-Completeness*, W.H. Freeman and Company (1979).

UNIT 2 · COMPLEXITY

Structure	Page Nos.
2.0 Introduction	19
2.1 Objectives	22
2.2 Notations for the Growth Rates of Functions	22
2.3 Classification of Problems	32
2.4 Reduction, NP-Complete and NP-Hard Problems	37
2.5 Establishing NP-Completeness of Problems	38
2.6 Summary	42
2.7 Solutions/Answers	43
2.8 Further Readings	45

2.0 INTRODUCTION

In the previous unit, we introduced you to the fact that there are a large number of problems which cannot be solved by algorithmic means and discussed a number of issues about such problems.

The advantage of such a study is our becoming aware of the fact that in stead of attempting to write an algorithm for every problem that we are required to solve using a computer, we should first study the *essential nature* of the problem. In case the problem under consideration is not solvable by algorithmic means, we may accept other computational techniques including use of heuristics, numerical and/or statistical techniques. Even out of problems, which though theoretically have algorithmic solutions, yet require such large amount of resources, that this type of problems are designated as *infeasible* for the purpose of computational solution. Out of the problems, which are feasibly solvable, there are problems each of which may have more than one algorithms to solve the problem. For us, it is desirable to know which one is better among the available ones. For example, we can use the algorithms viz, Bubble sort, Insertion sort, Heapsort and Quicksort, for sorting a list of numbers. Their designs are different but the outcome is the same for all, for a given list of numbers. As, there are more than one algorithms available to us to sort a list of numbers, it is natural for us to think of using the algorithm which solves a particular sorting problem, in some way *better* than the others. In context of practical disciplines like computer applications, an *efficient* solution is generally taken as a *better* solution. Efficiency of an algorithm can be considered in terms of the efficient use of computer resources, such as processor time and memory space used. In addition to the efficiency of execution of algorithms, other factors like *time* (taken by a team of software engineers and/or programmers) *required for developing algorithms* and *reliability* may also be taken into consideration as factors towards overall efficiency of an algorithm.

However, most of the time, in respect of efficiency of algorithms, we are only concerned with the time and space requirements of execution of algorithms.

In this unit, we will discuss the *issue* of efficiency of computation of an algorithm in terms of the *amount of time used in its execution*. On the basis of analysis of an algorithm, the amount of time that is estimated to be required in executing an algorithm, will be referred to as the *time complexity* of the algorithm. The time complexity of an algorithm is measured in terms of some (basic) *time unit* (not second or nano-second). Generally, time taken in executing one move of a TM, is taken as (basic) time unit for the purpose. Or, alternatively, time taken in executing some elementary operation like addition, is taken as one unit. More complex operations like

Meanwhile, we have actually succeeded in making our discipline science, and in a remarkably simple way merely by deciding to call it "computer science"...

... we have seen computer programming is an art, because it applies accumulated knowledge to the work because it requires skill and ingenuity, and specially because it produces objects of beauty...

Donald E. Knut
in
Turing Award
Lecture (1974)

multiplication etc, are assumed to require an *integral* number of basic units. As mentioned earlier, given many algorithms (solutions) for solving a problem, we would like to choose the most efficient algorithm from amongst the available ones. For *comparing efficiencies of algorithms*, that solve a particular problem, *time complexities of algorithms* are considered as *functions of the sizes of the problems* (to be discussed). The *time complexity functions of the algorithms are compared in terms of their growth rates (to be defined)* as growth rates are considered important measures of *comparative efficiencies*.

The concept of the **size of a problem**, though a fundamental one, yet is difficult to define precisely. Generally, the size of a *problem*, is measured in terms of the size of the *input*. The concept of the size of an input of a problem may be explained informally through examples. In the case of multiplication of two $n \times n$ (squares) matrices, the size of the problem may be taken as n^2 , i.e, the number of elements in each matrix to be multiplied. For problems involving polynomials, the degrees of the polynomials may be taken as measure of the sizes of the problems.

Also, we may have an intuitive idea about the term **growth rate and its significance** in the comparative study of algorithms that can be designed to solve problems. For the time being, in stead of attempting a formal definition, we illustrate the concept of *growth rate of time complexity function* of an algorithm and its significance through the following example.

Let us consider two algorithms to solve a problem P, having time-complexities respectively as $f_1(n) = 1000n^2$ and $f_2(n) = 5n^4$, where size of the problem is assumed to be n . Then

$$f_1(n) \geq f_2(n) \quad \text{for } n \leq 14 \quad \text{and}$$

$$f_1(n) \leq f_2(n) \quad \text{for } n \geq 15.$$

Also, the increase in the ratio $(f_2(n)/f_1(n))$ is *faster* than increase in n . Thus, informally, growth rate of $f_2(n)$ is more than the growth rate of $f_1(n)$. In one sense, the algorithm having time complexity $f_2(n)$ is *inferior* to the algorithm having time complexity $f_1(n)$ as growth rate of $f_2(n)$ is *faster* than that of $f_1(n)$.

A number of well-known notations for the formal treatment of the growth rate will be introduced later on within this section itself.

For a problem, a solution with time complexity which can be expressed as a polynomial of the size of the problem, is considered to have an **efficient solution**. Unfortunately, not many problems that arise in practice, admit any efficient algorithms, as these problems can be solved, if at all, by only non-polynomial time algorithms. A problem which does not have any (*known*) polynomial time algorithm is called an **intractable problem**.

At this stage, it is important to be aware of the following relevant facts

- (i) **A non-polynomial function need not always be exponential:** For example, the function $f(n) = n \log_2 n$ is neither polynomial function nor exponential function of n , but, somewhere between the two*.
- (ii) **The term *solution* in its general form: need not be an algorithm.** If by tossing a coin, we get the correct answer to each instance of a problem, then the process of tossing the coin and getting answers constitutes a solution. But, the process is not an algorithm. Similarly, we solve problems based on **heuristics**, i.e, good

* For details, refer Page 415, Introduction to Automata Theory, Languages, and Computation (Second Edition) by Hopcroft, Motwani &Ullman, Pearson Education Inc (2001).

guesses which, generally but not necessarily always, lead to solutions. All such cases of solutions are not algorithms, or algorithmic solutions. To be more explicit, by an algorithmic solution A of a problem L (considered as a language) from a problem domain Σ^* , we mean that among other conditions, the following are satisfied:

- (a) A is a step-by-step method in which for each instance of the problem, there is a definite sequence of execution steps (not involving any guess work).
- (b) A terminates for each $x \in \Sigma^*$, irrespective of whether $x \in L$ or $x \notin L$.

In this sense of algorithmic solution, only a solution by a Deterministic TM is called an algorithm. A solution by a Non-Deterministic TM may not be an algorithm.

- (iii) However, for every NTM solution, there is a Deterministic TM (DTM) solution of a problem. Therefore, if there is an NTM solution of a problem, then there is an algorithmic solution of the problem. However, the symmetry may end here.

The computational equivalence of Deterministic and Non-Deterministic TMs does not state or guarantee any equivalence in respect of requirement of resources like time and space by the Deterministic and Non-Deterministic models of TM, for solving a (solvable) problem. To be more precise, if a problem is solvable in polynomial-time by a Non-Deterministic Turing Machine, then it is, of course, guaranteed that there is a deterministic TM that solves the problem, but it is not guaranteed that there exists a Deterministic TM that solves the problem in polynomial time. Rather, this fact forms the basis for one of the deepest open questions of Mathematics, which is stated as 'whether $P = NP$ ' (P and NP to be defined soon).

The question put in simpler language means: Is it possible to design a Deterministic TM to solve a problem in polynomial time, for which, a Non-Deterministic TM that solves the problem in polynomial time, has already been designed?

We summarize the above discussion from the intractable problem's definition onward. Let us begin with definitions of the notions of P and NP .

P denotes the class of all problems, for each of which there is at least one known polynomial time Deterministic TM solving it.

NP denotes the class of all problems, for each of which, there is at least one known Non-Deterministic polynomial time solution. However, this solution may not be reducible to a polynomial time algorithm, i.e., to a polynomial time DTM.

Thus starting with two distinct classes of problems, viz, tractable problems and intractable problems, we introduced two classes of problems called P and NP . Some interesting relations known about these classes are:

- (i) $P =$ set of tractable problems
- (ii) $P \subseteq NP$.

(The relation (ii) above simply follows from the fact that every Deterministic TM is a special case of a Non-Deterministic TM).

However, it is not known whether $P=NP$ or $P \subset NP$. This forms the basis for the subject matter of the rest of the chapter. As a first step, we introduce some notations to facilitate the discussion of the concept of computational complexity.

2.1. OBJECTIVES

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

- explain the concepts of time complexity, size of a problem, growth rate of a function;
- define and explain the well-known notations for growth rates of functions, viz O , Ω , Θ , o , ω ;
- explain criteria for classification of problems into undefinable defineable but not solvable, solvable but not feasible, P, NP, NP-hard and NP-Complete etc.;
- define a number of problems which are known to be NP-complete problems;
- explain polynomial-reduction as a technique of establishing problems as NP-hard;
- establish NP-completeness of a number of problems.

2.2 NOTATIONS FOR GROWTH RATES OF FUNCTIONS

2.2.1 The Constant Factor in Complexity Measure

The time required by a solution or an algorithm for solving a (solvable) problem, depends *not only* on the size of the problem/input and the number of operations that the algorithm/solution uses, *but also* on the hardware and software used to execute the solution. However, the effect of change/improvement in hardware and software on the time required may be closely approximated by a *constant*.

Suppose, a supercomputer executes *instructions* one million times faster than another computer. Then irrespective of the size of a (solvable) problem and the solution used to solve it, the supercomputer solves the *problem* roughly million times faster than the computer, if the same solution is used on both the machines to solve the problem. Thus we conclude that the time requirement for execution of a solution, changes roughly by a *constant factor* on change in hardware, software and environmental factors.

An important consequence of the above discussion is that if the time taken by one machine in executing a solution of a problem is a polynomial (or exponential) function in the size of the problem, then time taken by every machine is a polynomial (or exponential) function respectively, in the size of the problem. *Thus, functions differing from each other by constant factors, when treated as time complexities should not be treated as different, i.e., should be treated as complexity-wise equivalent.*

2.2.2 Asymptotic Considerations

Computers are generally used to solve problems involving *complex* solutions. The complexity of solutions may be either because of the large number of involved computational steps and/or large size of input data. The plausibility of the claim apparently follows from the fact that, when required, computers are used *generally not to find the product of two 2×2 matrices but to find the product of two $n \times n$ matrices* for large n running into hundreds or even thousands.

Similarly, computers, when required, are generally used *not to find roots of quadratic equations but for finding roots of complex equations including polynomial equations of degrees more than hundreds or sometimes even thousands.*

The above discussion leads to the conclusion that when considering time complexities $f_1(n)$ and $f_2(n)$ of (computer) solutions of a problem of size n , we need to consider and compare the behaviors of the two functions only for large values of n . If the relative behaviors of two functions for smaller values conflict with the relative behaviours for larger values, then we may ignore the conflicting behaviour for smaller values. For example, if the earlier considered two functions

$$\begin{aligned} f_1(n) &= 1000 n^2 & \text{and} \\ f_2(n) &= 5n^4 \end{aligned}$$

represent time complexities of two solutions of a problem of size n , then despite the fact that

$$f_1(n) \geq f_2(n) \quad \text{for } n \leq 14,$$

we would still prefer the solution having $f_1(n)$ as time complexity because

$$f_1(n) \leq f_2(n) \quad \text{for all } n \geq 15.$$

This explains the reason for the presence of the phrase ' $n \geq k$ ' in the definitions of the various measures of complexities discussed below:

2.2.3 Well Known Asymptotic Growth Rate Notations

In the following we discuss some well-known growth rate notations. These notations denote relations from functions to functions.

For example, if functions

$f, g: \mathbb{N} \rightarrow \mathbb{N}$ are given by

$$f(n) = n^2 - 5n \quad \text{and}$$

$$g(n) = n^2$$

then

$$O(f(n)) = g(n) \quad \text{or} \quad O(n^2 - 5n) = n^2$$

(the notation O to be defined soon).

To be more precise, each of these notations is a mapping that associates a set of functions to each function. For example, if $f(n)$ is a polynomial of degree k then the set $O(f(n))$ includes all polynomials of degree less than or equal to k .

The five well-known notations and how these are pronounced:

- (i) $O(O(n^2))$ is pronounced as 'big-oh of n^2 ' or sometimes just as 'oh of n^2 '
- (ii) $\Omega(\Omega(n^2))$ is pronounced as 'big-omega of n^2 ' or sometimes just as 'omega of n^2 '
- (iii) $\Theta(\Theta(n^2))$ is pronounced as 'theta of n^2 '
- (iv) $o(o(n^2))$ is pronounced as 'little-oh of n^2 '
- (v) $\omega(\omega(n^2))$ is pronounced as 'little-omega of n^2 '

Remark 2.2.3.1

In the discussion of any one of the five notations, generally two functions say f and g are involved. The functions have their domains and Codomains as \mathbb{N} , the set of natural numbers, i.e,

$$\begin{aligned} f: \mathbb{N} &\rightarrow \mathbb{N} \\ g: \mathbb{N} &\rightarrow \mathbb{N} \end{aligned}$$

These functions may also be considered as having domain and codomain as \mathbb{R} .

Remark 2.2.3.2

The purpose of these asymptotic growth rate notations and functions denoted by these notations, is to facilitate the recognition of *essential character of a complexity function* through some simpler functions delivered by these notations. For example, a complexity function $f(n) = 5004 n^3 + 83 n^2 + 19 n + 408$, has essentially the same behaviour as that of $g(n) = n^3$ as the problem size n becomes larger and larger. But $g(n) = n^3$ is much more comprehensible than the function $f(n)$. Let us discuss the notations, starting with the notation O .

2.2.4 The Notation O

Provides asymptotic *upper bound* for a given function. Let $f(x)$ and $g(x)$ be two functions each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then $f(x)$ is said to be $O(g(x))$ (*pronounced as big-oh of g of x*) if there exist two positive integer/real number Constants C and k such that

$$f(x) \leq C g(x) \quad \text{for all } x \geq k \tag{A}$$

(The restriction of being positive on integers/ reals is justified as all complexities are positive numbers)

Example 2.2.4.1: For the function defined by

$$f(x) = 2x^3 + 3x^2 + 1$$

show that

- (i) $f(x) = O(x^3)$
- (ii) $f(x) = O(x^4)$
- (iii) $x^3 = O(f(x))$
- (iv) $x^4 \neq O(f(x))$
- (v) $f(x) \neq O(x^2)$

Solutions

Part (i)

Consider

$$\begin{aligned} f(x) &= 2x^3 + 3x^2 + 1 \\ &\leq 2x^3 + 3x^3 + 1 \quad x^3 = 6x^3 \end{aligned} \quad \text{for all } x \geq 1$$

(by replacing each term x^2 by the highest degree term x^3)

\therefore there exist $C = 6$ and $k = 1$ such that $f(x) \leq C \cdot x^3$ for all $x \geq k$

Thus we have found the required constants C and k . Hence $f(x)$ is $O(x^3)$.

Part (ii)

As above, we can show that

$$f(x) \leq 6x^4 \quad \text{for all } x \geq 1.$$

However, we may also, by computing some values of $f(x)$ and x^4 , find C and k as follows:

$$\begin{array}{ll} f(1) = 2+3+1 = 6 & ; \quad (1)^4 = 1 \\ f(2) = 2 \cdot 2^3 + 3 \cdot 2^2 + 1 = 29 & ; \quad (2)^4 = 16 \\ f(3) = 2 \cdot 3^3 + 3 \cdot 3^2 + 1 = 82 & ; \quad (3)^4 = 81 \end{array}$$

for $C = 2$ and $k = 3$ we have
 $f(x) \leq 2x^4$ for all $x \geq k$

Hence $f(x)$ is $O(x^4)$.

Part (iii)

for $C = 1$ and $k = 1$ we get
 $x^3 \leq C(2x^3 + 3x^2 + 1)$ for all $x \geq k$

Part (iv)

We prove the result by contradiction. Let there exist positive constants C and k such that

$$\begin{aligned} x^4 &\leq C(2x^3 + 3x^2 + 1) \quad \text{for all } x \geq k \\ \therefore x^4 &\leq C(2x^3 + 3x^3 + x^3) = 6Cx^3 \quad \text{for } x \geq k \\ \therefore x^4 &\leq 6Cx^3 \quad \text{for all } x \geq k. \end{aligned}$$

implying $x \leq 6C$ for all $x \geq k$

But for $x = \max\{6C + 1, k\}$, the previous statement is not true.

Hence the proof.

Part (v)

Again we establish the result by contradiction.

$$\text{Let } O(2x^3 + 3x^2 + 1) = x^2$$

Then for some positive numbers C and k

$$2x^3 + 3x^2 + 1 \leq Cx^2 \quad \text{for all } x \geq k,$$

implying

$$x^3 \leq Cx^2 \quad \text{for all } x \geq k \quad (\because x^3 \leq 2x^3 + 3x^2 + 1 \quad \text{for all } x \geq 1)$$

implying

$$x \leq C \quad \text{for } x \geq k$$

Again for $x = \max\{C + 1, k\}$

The last inequality does not hold. Hence the result.

Example: The big-oh notation can be used to estimate S_n , the sum of first n positive integers

Hint: $S_n = 1+2+3+\dots+n \leq n+n+\dots+n = n^2$
 Therefore, $S_n = O(n^2)$.

Remark 2.2.4.2

It can be easily seen that for given functions $f(x)$ and $g(x)$, if there exists one pair of and k with $f(x) \leq C.g(x)$ for all $x \geq k$, then there exist infinitely many pairs (C_i, k_i) which satisfy

$$f(x) \leq C_i g(x) \quad \text{for all } x \geq k_i.$$

Because for any $C_i \geq C$ and any $k_i \geq k$, the above inequality is true., if $f(x) \leq c.g(x)$ for all $x \geq k$.

2.2.5 The Ω Notation

Provides an asymptotic *lower bound* for a given function.

Let $f(x)$ and $g(x)$ be two functions, each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then $f(x)$ is said to be $\Omega(g(x))$ (*pronounced as big-omega of g of x*) if there exist two positive integer/real number Constants C and k such that

$$f(x) \geq C(g(x)) \quad \text{whenever } x \geq k$$

Example 2.2.5.1: For the functions

$$f(x) = 2x^3 + 3x^2 + 1 \text{ and } h(x) = 2x^3 - 3x^2 + 2$$

show that

- (i) $f(x) = \Omega(x^3)$
- (ii) $h(x) = \Omega(x^3)$
- (iii) $h(x) = \Omega(x^2)$
- (iv) $x^3 = \Omega(h(x))$
- (v) $x^2 \neq \Omega(h(x))$

Solutions:

Part (i)

For $C=1$, we have

$$f(x) \geq C x^3 \text{ for all } x \geq 1$$

Part (ii)

$$h(x) = 2x^3 - 3x^2 + 2$$

Let C and $k > 0$ be such that

$$2x^3 - 3x^2 + 2 \geq C x^3 \text{ for all } x \geq k$$

$$\text{i.e } (2-C) x^3 - 3x^2 + 2 \geq 0 \text{ for all } x \geq k$$

Then $C = 1$ and $k \geq 3$ satisfy the last inequality.

Part (iii)

$$2x^3 - 3x^2 + 2 = \Omega(x^2)$$

Let the above equation be true.

Then there exists positive numbers C and k

s.t.

$$2x^3 - 3x^2 + 2 > C x^2 \text{ for all } x \geq k$$

$$2x^3 - (3+C)x^2 + 2 \geq 0$$

It can be easily seen that lesser the value of C , better the chances of the above inequality being true. So, to begin with, let us take $C = 1$ and try to find a value of k s.t

$$2x^3 - 4x^2 + 2 \geq 0.$$

For $x \geq 2$, the above inequality holds
 $\therefore k=2$ is such that

$$2x^3 - 4x^2 + 2 \geq 0 \text{ for all } x \geq k$$

Part (iv)

Let the equality

$$x^3 = \Omega(2x^3 - 3x^2 + 2)$$

be true. Therefore, let $C > 0$ and $k > 0$ be such that

$$x^3 \geq C(2x^3 - 3/2 x^2 + 1)$$

For $C = 1/4$ and $k = 1$, the above inequality is true.

Part (v)

We prove the result by contradiction.

$$\text{Let } x^2 = \Omega(3x^3 - 2x^2 + 2)$$

Then, there exist positive constants C and k such that

$$x^2 \geq C(3x^3 - 2x^2 + 2) \text{ for all } x \geq k$$

$$\text{i.e. } (2C+1)x^2 \geq 3Cx^3 + 2 \geq Cx^3 \text{ for all } x \geq k$$

$$\frac{2C+1}{C} \geq x \text{ for all } x \geq k$$

$$\text{But for any } x \geq 2 \frac{(2C+1)}{C},$$

The above inequality can not hold. Hence contradiction.

2.2.6 The Notation Θ

Provides simultaneously *both* asymptotic *lower* bound and asymptotic *upper* bound for a given function.

Let $f(x)$ and $g(x)$ be two functions, each from the set of natural numbers or positive real numbers to positive real numbers. Then $f(x)$ said to be $\Theta(g(x))$ (*pronounced as big-theta of g of x*) if, there exist positive constants C_1 , C_2 and k such that $C_2 g(x) \leq f(x) \leq C_1 g(x)$ for all $x \geq k$.

(Note the last inequalities represent two conditions to be satisfied simultaneously viz $C_2 g(x) \leq f(x)$ and $f(x) \leq C_1 g(x)$)

We state the following theorem without proof, which relates the three functions O , Ω , Θ

Theorem: For any two functions $f(x)$ and $g(x)$, $f(x) = \Theta(g(x))$ if and only if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.

Examples 2.2.6.1: For the function $f(x) = 2x^3 + 3x^2 + 1$, show that

(i) $f(x) = \Theta(x^3)$

(ii) $f(x) \neq \Theta(x^2)$

(iii) $f(x) \neq \Theta(x^4)$

Solutions

Part (i)

for $C_1 = 3, C_2 = 1$ and $k = 4$

I. $C_2 x^3 \leq f(x) \leq C_1 x^3$ for all $x \geq k$

Part (ii)

We can show by contradiction that no C_1 exists.

Let, if possible for some positive integers k and C_1 , we have $2x^3 + 3x^2 + 1 \leq C_1 x^2$ for all $x \geq k$

Then

$$x^3 \leq C_1 x^2 \text{ for all } x \geq k$$

i.e,

$$x \leq C_1 \text{ for all } x \geq k$$

But for

$$x = \max \{C_1 + 1, k\}$$

The last inequality is not true

Part (iii)

$$f(x) \neq \Theta(x^4)$$

We can show by contradiction that there does not exist C_2 s.t

$$C_2 x^4 \leq (2x^3 + 3x^2 + 1)$$

If such a C_2 exists for some k then $C_2 x^4 \leq 2x^3 + 3x^2 + 1 \leq 6x^3$ for all $x \geq k \geq 1$,

implying

$$C_2 x \leq 6 \text{ for all } x \geq k$$

But for $x = \left(\frac{6}{C_2} + 1\right)$

the above inequality is false. Hence, proof of the claim by contradiction.

2.2.7 The Notation \mathcal{O}

The asymptotic upper bound provided by big-oh notation may or may not be tight in the sense that if $f(x) = 2x^3 + 3x^2 + 1$

Then for $f(x) = \mathcal{O}(x^3)$, though there exist C and k such that $f(x) \leq C(x^3)$ for all $x \geq k$

The two-edge set $\{(1, 4), (2, 3)\}$ is an edge cover for the graph.

Problem 13: **Exact cover problem:** For a given set $\mathcal{P} = \{S_1, S_2, \dots, S_k\}$, where each S_i is a subset of a given set S , is there a subset Q of \mathcal{P} such that for each x in S , there is exactly one S_i in Q for which x is in S_i ?

Example: Let $S = \{1, 2, \dots, 10\}$
and $\mathcal{P} = \{S_1, S_2, S_3, S_4, S_5\}$ s.t

$S_1 =$	$\{1, 3, 5\}$
$S_2 =$	$\{2, 4, 6\}$
$S_3 =$	$\{1, 2, 3, 4\}$
$S_4 =$	$\{5, 6, 7, 9, 10\}$
$S_5 =$	$\{7, 8, 9, 10\}$

Then $Q = \{S_1, S_2, S_3\}$ is a set cover for S .

Problem 14: **The knapsack problem:** Given a list of k integers n_1, n_2, \dots, n_k , can we partition these integers into two sets, such that sum of integers in each of the two sets is equal to the same integer?

2.4 REDUCTION, NP-COMPLETE AND NP-HARD PROBLEMS

Earlier we (informally) explained that a problem is called NP-Complete if P has at least one Non-Deterministic polynomial-time solution and further, so far, no polynomial-time Deterministic TM is known that solves the problem.

In this section, we formally define the concept and then describe a general technique of establishing the NP-Completeness of problems and finally apply the technique to show some of the problems as NP-complete. We have already explained how a problem can be thought of as a language L over some alphabet Σ . Thus the terms *problem and language* may be interchangeably used.

For the formal definition of NP-completeness, *polynomial-time reduction*, as defined below, plays a very important role.

In the previous unit, we discussed *reduction technique* to establish some of the problems as undecidable. The method that was used for establishing undecidability of a language using the technique of reduction, may be briefly described as follows:

Let P_1 be a problem which is already known to be undecidable. We want to check whether a problem P_2 is undecidable or not. If we are able to design an algorithm which transforms or constructs an instance of P_2 for each instance of P_1 , then P_2 is also undecidable.

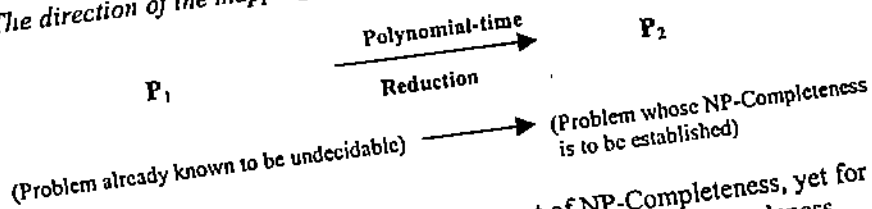
The process of transformation of the instances of the problem already known to the undecidable to instances of the problem, the undecidability is to be checked, is called *reduction*.

More or less similar, but, slightly different, rather special, reduction called *polynomial-time reduction* is used to establish NP-Completeness of problems.

Polynomial-time reduction is a polynomial-time algorithm which constructs the instances of a problem P_2 from the instances of some other problems P_1 .

... establishing the NP-Completeness (to be formally defined later) of a
 ... constitutes of designing a polynomial time reduction that constructs
 P_2 for each instance of P_1 , where P_1 is already known to be

The direction of the mapping must be clearly understood as shown below.



Though we have already explained the concept of NP-Completeness, yet for the sake of completeness, we give below the formal definition of NP-Completeness

Definition: NP-Complete Problem: A Problem P or equivalently its language L_1 is said to be NP-complete if the following two conditions are satisfied:

- (i) The problem L_2 is in the class NP
- (ii) For any problem L_2 in NP, there is a polynomial-time reduction of L_1 to L_2

In this context, we introduce below another closely related and useful concept.

Definition: NP-Hard Problem A problem L is said to be NP-hard if for any problem L_1 in NP, there is a polynomial-time reduction of L_1 to L

In other words, a problem L is hard if only condition (ii) of NP-Completeness is satisfied. But the problem has may be so hard that establishing L as an NP-class problem is so far not possible.

However, from the above definitions, it is clear that every NP-complete problem L must be NP-Hard and additionally should satisfy the condition that L is an NP-class problem.

In the next section, we discuss NP-completeness of some of problems discussed in the previous section.

2.5 ESTABLISHING NP-COMPLETENESS OF PROBLEMS

In general, the process of establishing a problem as NP-Complete is a two-step process. The first step, which in most of the cases is quite simple, constitutes of guessing possible solutions of the instances, one instance at a time, of the problem and then verifying whether the guess actually is a solution or not.

The second step involves designing a polynomial-time algorithm which reduces instances of an already known NP-Complete problem to instances of the problem, which is intended to be shown as NP-Complete.

However, to begin with, there is a major hurdle in execution of the second step. The above technique of reduction can not be applied unless we already have established at least one problem as NP-Complete. Therefore, for the first NP-Complete problem, the NP-Completeness has to be established in a different manner.

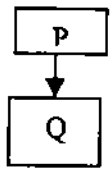
As mentioned earlier, Stephen Cook (1971) established Satisfiability as the first

NP-Complete problem. The proof was based on explicit reduction of the language of any non-deterministic, polynomial-time TM to the satisfiability problem.

The proof of Satisfiability problem as the first NP-Complete problem, is quite lengthy and we skip the proof. Interested readers may consult any of the text given in the reference.

Assuming the satisfiability problem as NP-complete, the rest of the problems that we establish as NP-complete, are established by reduction method as explained above.

A diagrammatic notation of the form



Indicates: Assuming P is already established as NP-Complete, the NP-Completeness of Q is established by through a polynomial-time reduction from P to Q

A scheme for establishing NP-Completeness of some the problems mentioned in Section 2.2, is suggested by Fig. 2.1 given below

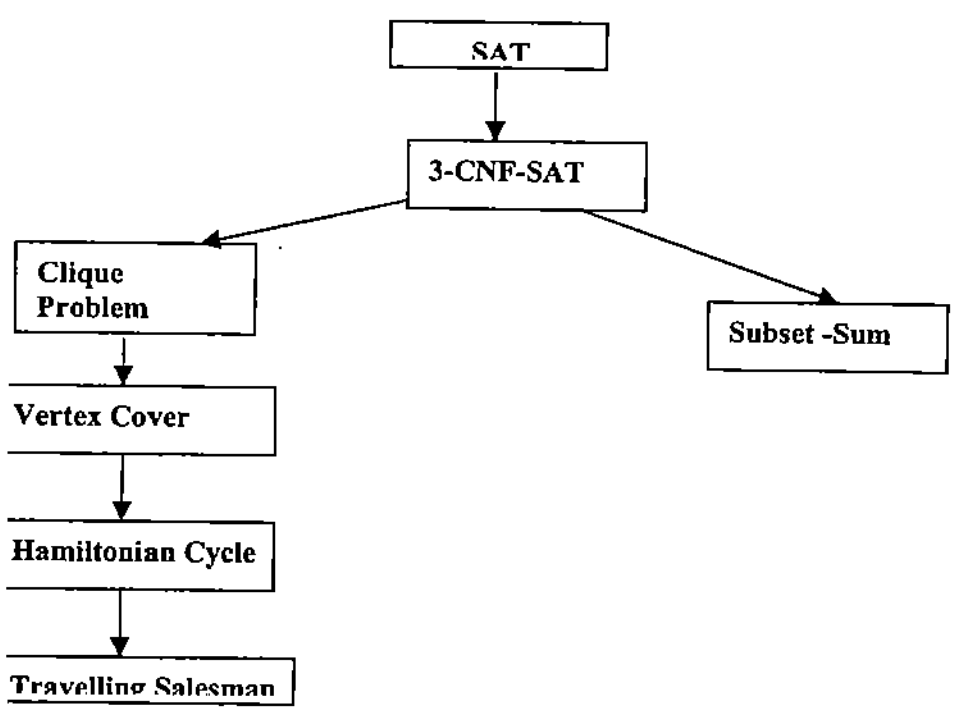


Fig. 2.1

Example 2.4.1: Show that the Clique problem is an NP-complete problem.

Proof : The verification of whether every pairs of vertices is connected by an edge in E, is done for different pairs of vertices by a Non-deterministic TM, i.e, in parallel. Hence, it takes only polynomial time because for each of n vertices we need to verify at most $n(n+1)/2$ edges, the maximum number of edges in a graph with n vertices.

We next show that 3- CNF-SAT problem can be transformed to clique problem in polynomial time.

Take an instance of 3-CNF-SAT. An instance of 3CNF-SAT consists of a set of n clauses, each consisting of exactly 3 literals, each being either a variable or negated variable. It is satisfiable if we can choose literals in such a way that:

- at least one literal from each clause is chosen
- if literal of form x is chosen, no literal of form $\neg x$ is considered.

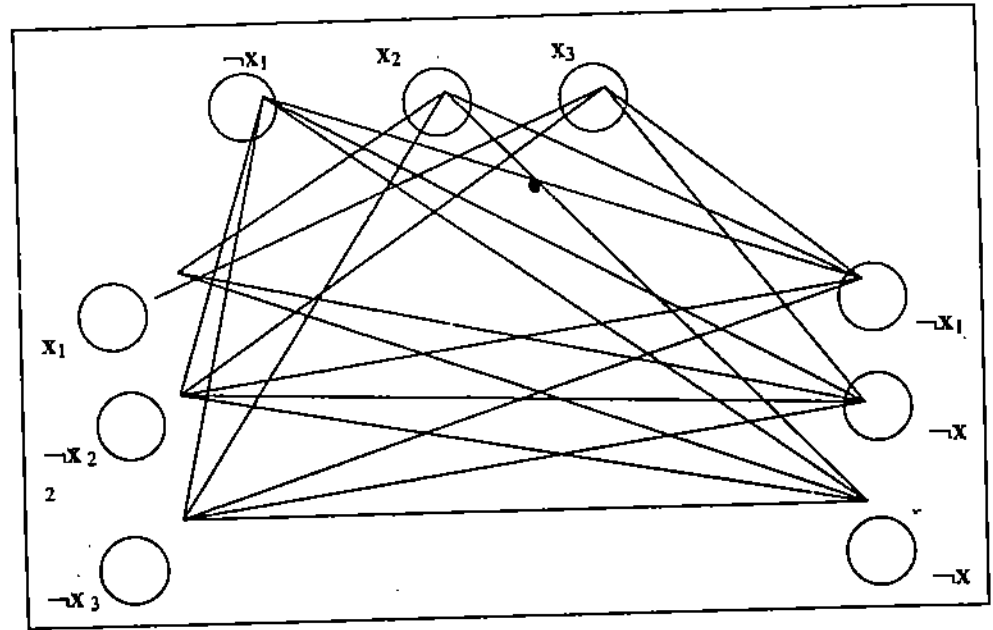


Fig. 2.2

For each of the literals, create a graph node, and connect each node to every node in other clauses, except those with the same variable but different sign. This graph can be easily computed from a boolean formula ϕ in 3-CNF-SAT in polynomial time. Consider an example, if we have

$$\phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

then G is the graph shown in Figure 2.2 above.

In the given example, a satisfying assignment of ϕ is $(x_1 = 0, x_2 = 0, x_3 = 1)$. A corresponding clique of size $k = 3$ consists of the vertices corresponding to x_2 from the first clause, $\neg x_3$ from the second clause, and $\neg x_3$ from the third clause.

The problem of finding n -element clique is equivalent to finding a set of literals satisfying SAT. Because there are no edges between literals of the same clause, such a clique must contain exactly one literal from each clause. And because there are no edges between literals of the same variable but different sign, if node of literal x is in the clique, no node of literal of form $\neg x$ is.

This proves that finding n -element clique in $3n$ -element graph is NP-Complete.

Example 5: Show that the Vertex cover problem is an NP- complete.

A *vertex cover* of an undirected graph $G = (V, E)$ is a subset V' of the vertices of the graph which contains at least one of the two endpoints of each edge.

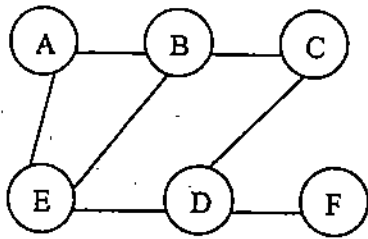


Fig. 2.3

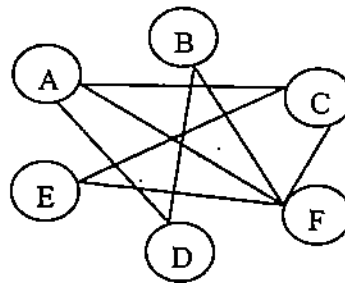


Fig. 2.4

The vertex cover problem is the optimization problem of finding a vertex cover of minimum size in a graph. The problem can also be stated as a decision problem :

VERTEX-COVER = $\{ \langle G, k \rangle \mid \text{graph } G \text{ has a vertex cover of size } k \}$.

A deterministic algorithm to find a vertex cover in a graph is to list all subsets of vertices of size k and check each one to see whether it forms a vertex cover. This algorithm is exponential in k .

Proof : To show that Vertex cover problem \in NP, for a given graph $G = (V, E)$, we take $V' \subseteq V$ and verifies to see if it forms a vertex cover. Verification can be done by checking for each edge $(u, v) \in E$ whether $u \in V'$ or $v \in V'$. This verification can be done in polynomial time.

Now, We show that clique problem can be transformed to vertex cover problem in polynomial time. This transformation is based on the notion of the complement of a graph G . Given an undirected graph $G = (V, E)$, we define the complement of G as $G' = (V, E')$, where $E' = \{ (u, v) \mid (u, v) \notin E \}$. i.e G' is the graph containing exactly those edges that are not in G . The transformation takes a graph G and k of the clique problem. It computes the complement G' which can be done in polynomial time.

To complete the proof, we can show that this transformation is indeed reduction : the graph has a clique of size k if and only if the graph G' has a vertex cover of size $|V| - k$.

Suppose that G has a clique $V' \subseteq V$ with $|V'| = k$. We claim that $V - V'$ is a vertex cover in G' . Let (u, v) be any edge in E' . Then, $(u, v) \notin E$, which implies that atleast one of u or v does not belong to V' , since every pair of vertices in V' is connected by an edge of E . Equivalently, atleast one of u or v is in $V - V'$, which means that edge (u, v) is covered by $V - V'$. Since (u, v) was chosen arbitrarily from E' , every edge of E' is covered by a vertex in $V - V'$. Hence, the set $V - V'$, which has size $|V| - k$, forms a vertex cover for G' .

Conversely, suppose that G' has a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$. Then, for all $u, v \in V$, if $(u, v) \in E'$, then $u \in V'$ or $v \in V'$ or both. The contrapositive of this implication is that for all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$. In other words, $V - V'$ is a clique, and it has size $|V| - |V'| = k$.

For example, The graph $G(V, E)$ has a clique $\{A, B, E\}$ given by Figure 7.4. The complement of graph G is given by G' shown as Figure 7.5 and have independent set given by $\{C, D, F\}$.

This proves that finding the vertex cover is NP-Complete.

- Ex.5) Show that the k-colorability problem is NP.
- Ex.6) Show that the Independent Set problem is NP- complete.
- Ex.7) Show that the Travelling salesman problem is NP- complete.
-

2.6 SUMMARY

In this unit in number of concepts are defined.

P denotes the class of all problems, for each of which there is at least one *known* polynomial time Deterministic TM solving it.

NP denotes the class of all problems, for each of which, there is at least one known Non-Deterministic polynomial time solution. However, this solution may not be reducible to a polynomial time algorithm, i.e, to a polynomial time DTM.

Next, five Well Known Asymptotic Growth Rate Notations are defined.

The notation **O** provides asymptotic *upper bound* for a given function. Let $f(x)$ and $g(x)$ be two functions each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then $f(x)$ is said to be $O(g(x))$ (*pronounced as big-oh of g of x*) if there exist two positive integer/real number Constants C and k such that

$$f(x) \leq C g(x) \quad \text{for all } x \geq k$$

The Ω notation provides an asymptotic *lower bound* for a given function

Let $f(x)$ and $g(x)$ be two functions, each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then $f(x)$ is said to be $\Omega(g(x))$ (*pronounced as big-omega of g of x*) if there exist two positive integer/real number Constants C and k such that

$$f(x) \geq C(g(x)) \quad \text{whenever } x \geq k$$

The Notation Θ

Provides simultaneously *both* asymptotic *lower bound* and asymptotic *upper bound* for a given function.

Let $f(x)$ and $g(x)$ be two functions, each from the set of natural numbers or positive real numbers to positive real numbers. Then $f(x)$ said to be $\Theta(g(x))$ (*pronounced as big-theta of g of x*) if, there exist positive constants C_1, C_2 and k such that $C_2 g(x) \leq f(x) \leq C_1 g(x)$ for all $x \geq k$.

The Notation o

Let $f(x)$ and $g(x)$ be two functions, each from the set of natural numbers or positive real numbers to positive real numbers

Further, let $C > 0$ be any number, then $f(x) = o(g(x))$ (*pronounced as little oh of g of x*) if there exists natural number k satisfying

$$f(x) < C.g(x) \quad \text{for all } x \geq k \geq 1$$

The Notation ω

Again the asymptotic lower bound Ω may or may not be tight. However, the asymptotic bound ω cannot be tight. The formal definition of ω is follows:

Let $f(x)$ and $g(x)$ be two functions each from the set of natural numbers or the set of positive real numbers, to set of positive real numbers.

Further

Let $C > 0$ be any number, then

$$f(x) = \omega(g(x))$$

if there exist a positive integer k s.t

$$f(x) > C g(x) \quad \text{for all } x \geq k$$

In Section 2.2 in defined, 14 well known problems, which are known to be NP-Complete.

In Section 2.3 we defined the following concepts.

A Polynomial-time reduction is a polynomial-time algorithm which constructs the instances of a problem P_2 from the instances of some other problems P_1

Definition: NP-Complete Problem: A Problem P or equivalently its language L_1 is said to be NP-complete if the following two conditions are satisfied:

- (i) The problem L_2 is in the class NP
- (ii) For any problem L_2 in NP, there is a polynomial-time reduction of L_1 to L_2

Definition: NP-Hard Problem A problem L is said to be NP-hard if for any problem L_1 in NP, there is a polynomial-time reduction of L_1 to L

Finally in Section 2.4, we discussed how some of the problems defined in Section 2.2 are established as NP-Complete.

2.7 SOLUTIONS/ANSWERS

Exercise 1: $n!/n^n = (n/n) ((n-1)/n) ((n-2)/n) ((n-3)/n) \dots (2/n)(1/n)$
 $= 1(1-(1/n)) (1-(2/n)) (1-(3/n)) \dots (2/n)(1/n)$

Each factor on the right hand side is less than equal to 1 for all value of n . Hence, The right hand side expression is always less than one.

Therefore, $n!/n^n \leq 1$

or, $n! \leq n^n$

Therefore, $n! = O(n^n)$

Exercise 2: For large value of n , $3 \log n \ll n^2$

Therefore, $3 \log n / n^2 \ll 1$

$$(n^2 + 3 \log n) / n^2 = 1 + 3 \log n / n^2$$

$$\text{or, } (n^2 + 3 \log n) / n^2 < 2$$

$$\text{or, } n^2 + 3 \log n = O(n^2).$$

Exercise 3 : We have, $2^n/5^n < 1$
or, $2^n < 5^n$
Therefore, $2^n = O(5^n)$.

Exercise 4 : Given a set of integers, we have to divide the set in to two disjoint sets such that their sum value is equal .

A deterministic algorithm to find two disjoint sets is to list all possible combination of two subsets such that one set contain k elements and other contains remaining $(n-k)$ elements. Then to check if the sum of elements of one set is equal to the sum of elements of another set. Here, the possible number of combination is $C(n, k)$. This algorithm is exponential in n .

To show that the partition problem \in NP, for a given set S , we take $S_1 \subseteq S, S_2 \subseteq S$ and $S_1 \cap S_2 = \emptyset$ and verify to see if the sum of all elements of set S_1 is equal to the sum of all elements of set S_2 . This verification can be done in polynomial time. Hence, the partition problem is NP.

Exercise 5 : The graph coloring problem is to determine the minimum number of colors needed to color given graph $G(V, E)$ vertices such that no two adjacent vertices has the same color. A deterministic algorithm for this requires exponential time.

If we cast the graph-coloring problem as a decision problem *i.e.* Can we color the graph G with k -colors such that no two adjacent vertices have same color ? We can verify that if this is possible then it is possible in polynomial time.

Hence, The graph -coloring problem is NP.

Exercise 6 : An independent set is defined as a subset of a vertices in a graph such that no two vertices are adjacent.

The independent set problem is the optimization problem of finding an independent set of maximum size in a graph. The problem can also be stated as a decision problem :

INDEPENDENT-SET = $\{ \langle G, k \rangle \mid G \text{ has an independent set of atleast size } k \}$.

A deterministic algorithm to find an independent set in a graph is to list all subsets of vertices of size k and check each one to see whether it forms an independent set. This algorithm is exponential in k .

Proof : To show that the independent set problem \in NP, for a given graph $G = (V, E)$, we take $V' \subseteq V$ and verifies to see if it forms an independent set. Verification can be done by checking for $u \in V'$ and $v \in V'$, does $(u, v) \in E$. This verification can be done in polynomial time.

Now, We show that clique problem can be transformed to independent set problem in polynomial time. The transformation is similar clique to vertex cover. This transformation is based on the notion of the complement of a graph G . Given an undirected graph $G = (V, E)$, we define the complement of G as $G' = (V, E')$, where $E' = \{ (u, v) \mid (u, v) \notin E \}$. *i.e* G' is the graph containing exactly those edges that are not in G . The transformation takes a graph G and k of the clique problem. It computes the complement G' which can be done in polynomial time.

To complete the proof, we can show that this transformation is indeed reduction : the graph has a clique of size k if and only if the graph G' has an independent set of size $|V| - k$.

Suppose that G has a clique $V' \subseteq V$ with $|V'| = k$. We claim that $V - V'$ is an independent set in G' . Let (u, v) be any edge in E' . Then, $(u, v) \notin E$, which implies that at least one of u or v does not belong to V' , since every pair of vertices in V' is connected by an edge of E . Equivalently, at least one of u or v is in $V - V'$, which means that edge (u, v) is covered by $V - V'$. Since (u, v) was chosen arbitrarily from E' , every edge of E' is covered by a vertex in $V - V'$. So, either u or v is in $V - V'$ and no two adjacent vertices are in $V - V'$. Hence, the set $V - V'$, which has size $|V| - k$, forms an independent set for G' .

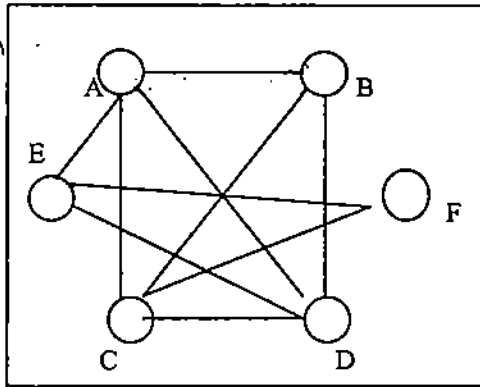


Fig. 2.5

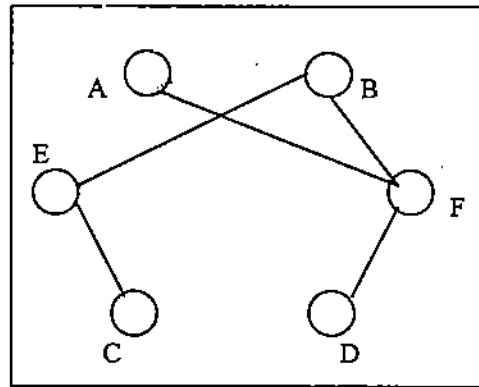


Fig. 2.6

For example, The graph $G(V, E)$ has a clique $\{A, B, C, D\}$ given by Figure 2.5. The complement of graph G is given by G' shown as Figure 2.6 and have independent set given by $\{E, F\}$

This transformation can be performed in polynomial time. This proves that finding the independent set problem is NP-Complete.

Exercise 7:

Proof : To show that travelling salesman problem \in NP, we show that verification of the problem can be done in polynomial time. Given a constant M and a closed circuit path of a weighted graph $G = (V, E)$. Does such path exists in graph G and total weight of such path is less than M ? Verification can be done by checking, does $(u, v) \in E$ and the sum of weights of these edges is less than M . This verification can be done in polynomial time.

Now, We show that Hamiltonian circuit problem can be transformed to travelling problem in polynomial time. It can be shown that, Hamiltonian circuit problem is a special case of the travelling salesman problem. Towards this goal, given any Graph $G(V, E)$, we construct an instance of the $|V|$ -city Travelling salesman by letting $d_{ij} = 1$ if $(v_i, v_j) \in E$, and 2 otherwise. We let the cost of travel M equal to $|V|$. It is immediate that there is a tour of length M or less if and only if there exists a Hamiltonian circuit in G .

Hence, The travelling salesman is NP-complete.

2.8 FURTHER READINGS

1. H.R. Lewis & C.H.Papadimitriou: *Elements of the Theory of computation*, PHI, (1981).
2. J.E. Hopcroft, R.Motwani & J.D.Ullman: *Introduction to Automata Theory, Languages, and Computation* (II Ed.) Pearson Education Asia (2001).

**Complexity and
Computability**

3. J.E. Hopcroft and J.D. Ullman: *Introduction to Automata Theory, Language, and Computation*, Narosa Publishing House (1987).
4. J.C. Martin: *Introduction to Languages and Theory of Computation*, Tata-Mc Graw-Hill (1997).
5. K.N. Rosen: *Discrete Mathematics and Its Applications (Fifth Edition)* Tata McGraw-Hill (2003).
6. T.H. Cormen, C.E. Leiserson & C. Stein: *Introduction to Algorithms (Second Edition)* Prentice – Hall of India (2002).

UNIT 3 APPLICATIONS

Structure	Page No.
3.0 Introduction	47
3.1 Objectives	47
3.2 Applications of Finite Automata	48
3.3 Applications of Regular Expression	51
3.4 Application of Context-Free Grammars	56
3.5 Summary	
3.6 Further Readings & References	60
Epilogue: For Those Who Care...	61

3.0 INTRODUCTION

In the paragraph by Prof. Scot, Thomas Shadwell satirically emphasizes, for all teachers and students, the need for conducting practicals and experiencing first-hand the phenomena, which forms the basis of the subject matter of our study. In the case of computer science, the above mentioned emphasis translates to the need for practising and experiencing first-hand the computational phenomena, by developing programs and carrying out projects, if not large ones, at least of moderate sizes.

In order to develop taste in students for practicals, it is necessary to emphasize applications in addition to the theoretical, mind-expanding exercises. Today ideas from Theoretical Computer Science (TCS) find applications in as varied disciplines as Mathematics, Astronomy, Manufacturing and Biology. Within Computer Science, the ideas from TCS are useful in various areas including cryptography and secure computation, VLSI design, and communication networks. A nice summary of the applications of TCS is given in *J.E. Savage, A.L. Selman and C. Smith: (2001)*.

In this unit, we discuss applications of Finite Automata to Web search and extraction of information from text; of Regular Expressions to designing of Lexical Analyzers; and of Context-Free Grammars (CFG) to designing of Parsers.

The elegant notation – now well-known as BNF – played a significant role in facilitating definitions of (context-free portions of) programming languages. The following references may be quite useful in this direction: J.W. Backus [1959], P. Naur et al. [1960].

The tools like LEX for lexical analysis and YACC for parsing are useful in developing student's interest in practical work.

For details of LEX and YACC, the following references may be useful: M.E. Lesk [1975], N. Chomsky [1956], S.C. Johnson [1975].

3.1 OBJECTIVES

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

- explain application of Finite Automata to searches of the World Wide Web and textual information bases;

*Those of us... may well be reminded of Sir Nicholas Gimcrack, hero of the play *The Virtuoso*. It was written in 1676 by Thomas Shadwell to poke a little at the remarkable experiments then being done before the Royal Society of London. At one point in the play, Sir Nicholas is discovered lying on a table trying to learn to swim by imitating the motions of a frog in a bowl of water. When asked whether he had ever practiced swimming in water, he replies that he hates water and would never go near it! "I content myself," he said, "with the speculative part of swimming; I care not for the practical. I seldom bring anything to use... Knowledge is the ultimate end".*

Dana S. Scott
in
Turing Award Lecture
(1976)

- explain applications of Regular Expressions to lexical analysis tasks and also to textual search and subsection jobs.
- explain how ideas from Context-Free Grammars and BNF notations can be used in defining part of programming languages and also in parsing jobs.

3.2 APPLICATIONS OF FINITE AUTOMATA

Problems involving searches of the (World Wide) Web and other on-line textual information bases, generally require finding all the documents that contain some or all of a given set of words. Let us call such words in a given set as *keywords*. The techniques used for searching documents for keywords mainly use either *inverted indexes* or *automata* for the purpose. Search engines generally use inverted indexes.

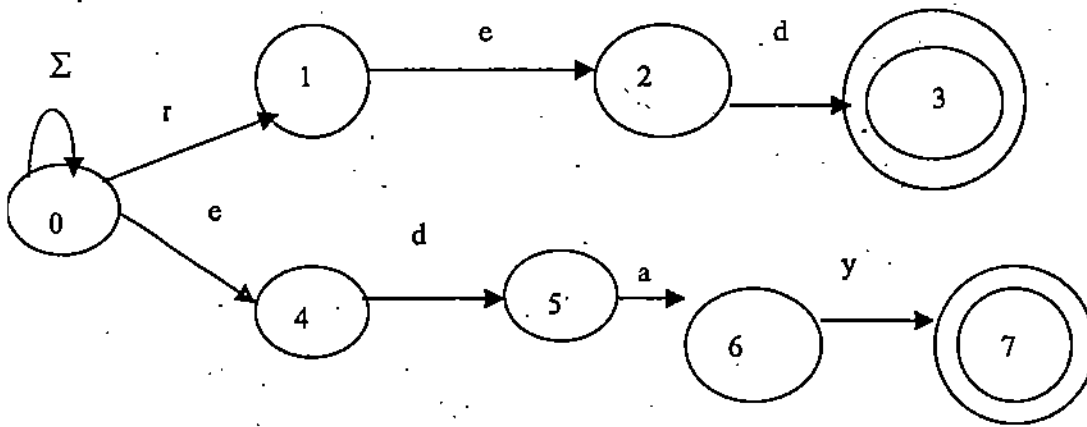
However, the automata-based searches are more suitable than inverted indexes in the type of information bases in which

- (i) most of the information contents change frequently as happens in the case of daily on-line news items, and
- (ii) the documents being searched are not, and even can not be, catalogued, which makes searches based on inverted indexes difficult.

In such situations, the task is accomplished using finite automata, through the following three-step sequence:

- (i) First, a Non-Deterministic Finite Automata (*NFA*) is designed that accepts and signals the acceptance of the keywords.
- (ii) Then, as an NFA is not a program, we use either of the following approaches for its implementation:
 - (a) Using subset construction, convert the NFA designed in Step (i) to an equivalent DFA and then simulate the DFA directly. The problem with this approach is that the number of states in the DFA increases exponentially with the increase in the number of states in the NFA
 - (b) Write a program that simulates the NFA by computing the set of states in which the NFA would be after reaching each input symbol. The advantage of the approach is that *without actually converting into a DFA*, we are able to simulate an equivalent DFA *which never has more states than the number of states in the NFA of Step (i)*. In a short while, we discuss the approach in detail.
- (iii) Writing an algorithm/program by simulating the DFA obtained at step (ii)
A mixture of the two approaches is actually used in the text-processing programs *egrep* and *fgrep*, which are advanced forms of UNIX *grep* program. **First, we illustrate the Step (i) of designing of NFA through an example.** Then we explain how to simulate a DFA equivalent to the NFA, as an illustration of Step (ii) (b)

Example: We design an NFA to recognize the strings *red* and *eday*. If Σ denotes the set of all possible printable ASCII characters, then the following diagram represents an NFA that recognizes the strings *red* and *eday*.



Now, we give below the details of the approach mentioned under Step (ii) b above and illustrate it with an example.

The states and transitions of the to-be-simulated DFA equivalent to the NFA of Step (i) are explained below. First, we discuss the states followed by the discussion for transitions. *It may be noted that each state of the simulated DFA is a set of states of the NFA designed in step (i) above. The states of the DFA are obtained from the states of NFA through the following two steps*

- (a) If q_0 is the start state of the NFA of Step (i) then $\{q_0\}$ is one of the states of the simulated DFA.
- (b) Other states of the DFA are constructed using *only those states p of the NFA which are reachable from the initial state q_0 of NFA through some input sequence $b_1 b_2 \dots b_m$. In other words, if δ is the transition relation of the NFA then p is the state which satisfies*

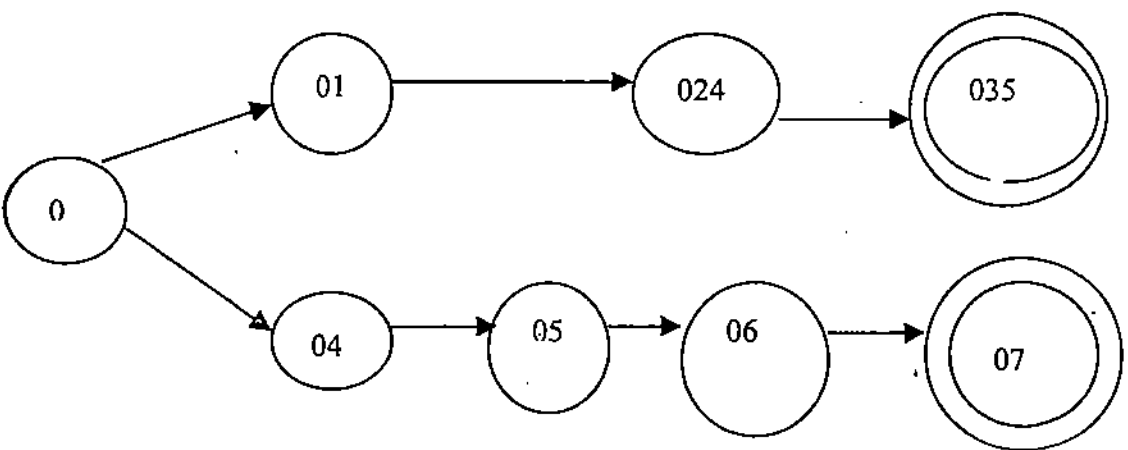
$$p \in \delta(q_0, b_1 b_2 \dots b_m)$$
for some input sequence $b_1 b_2 \dots b_m$.

Then, for each such state p of NFA, we obtain a state of the DFA as *the set of states of the NFA which consists of*

- (i) q_0 , the initial state of NFA
- (ii) p , the state which is reachable from q_0 through some sequence $b_1 b_2 \dots b_m$ of inputs and
- (iii) each of the other states of the NFA which is reachable from q_0 through a subsequence $b_j b_{j+1} \dots b_m$ for each $j = 2, \dots, m$, of the sequence $b_1 b_2 \dots b_m$.

Though we shall not attempt to prove that the above construction leads to an equivalent DFA having at most as many states as the NFA yet the fact is easily seen to be true from the following illustration.

The diagram *only for the states (transitions to be considered after a while)* of the simulated DFA corresponding to the NFA of the previous example, recognizing the strings *red* and *eday* is as given below



In the above diagram 04 denotes the state {0,4} and 035 denotes the state {0, 3, 5} etc. Next we discuss how the transitions between these states of the simulating DFA are obtained. As an illustration, first we explain the transitions between some states in the above diagram and then generalize.

Let us consider transitions out of the state 024 of the DFA. For this purpose we consider the two states viz 2 and 4 of the NFA. Then, for each input symbol out of the symbols constituting the strings red and eday, we find out the target states of the transitions from 2 and 4. First, we consider the symbol d. The state 2 in the NFA goes to state 3 on input of symbol d and state 4 goes to state 5 on input d. Taking the union of the target states alongwith initial state 0 we get {0, 3, 5} which corresponds to the state 035 of the DFA. Thus the state 024 goes to the state 035 on input d in the simulating DFA.

Next, we consider an input symbol say e for which there is no transition out of the two states 2 and 4 in the NFA. In such cases, we consider a transition out of the initial state 0. The state 0 goes to the state 4 on input of the symbol e in the NFA. Taking the state 4 alongwith the initial state 0 of the NFA, we get the set {0,4} which corresponds to the state 04 of the DFA. Thus the state 024 on input of the symbol e goes to the state 04 in the DFA. The above essentially explains the method of defining transitions of the proposed DFA. We summarise below the method of defining the transitions in general:

If q_0 is the initial state and p_1, p_2, \dots are the states of the NFA other than q_0 , then consider for each input symbol x , and each state $q_0 p_i p_j p_k$ of the DFA, the target states on input x of the transitions, in the NFA, from each of p_i, p_j etc which form a part of the name $q_0 p_i p_j p_k$ of the state of DFA. Collect all these target states.

If the set of target states is not empty, say it is $\{s_1, s_2, s_3\}$ then $q_0 s_1 s_2 s_3$ is the target state on input x from the state $q_0 p_i p_j p_k$. However, if the set of target states is empty then only we consider the transitions out of the initial state q_0 on input x . If s is the target state of this transition, then $q_0 s$ is the target state on input x from the state $q_0 p_i p_j p_k$.

The transition function δ for the simulating DFA is given by the following table, from which, if required, the transition diagram can be drawn.

	Input symbols	r	E	d	a	y	$\Sigma \sim \{e, r\}$	$\Sigma \sim \{a, e, r\}$	$\Sigma \sim \{d, e, r\}$	$\Sigma \sim \{e, r, y\}$
States	-	-	-	-	-	-	-	-	-	-
0	-	01	04	x	x	x	0	x	x	x
01	-	01	024	x	x	x	0	x	x	x
024	-	01	04	035	x	x	x	x	0	x
035	-	01	04	x	06	x	x	0	x	x
04	-	01	04	05	x	x	x	x	0	x
05	-	01	04	x	06	x	x	0	x	x
06	-	01	04	x	x	07	x	x	x	0
07	-	01	04	x	x	x	0	x	x	x

In the above table, the symbol 'x' denotes: *there is no transition for the corresponding (state, input) pair.*

3.3 APPLICATIONS OF REGULAR EXPRESSIONS

In the earlier units, we discussed how a particular type of languages (viz. regular languages) can be defined using finite automata. From the definition of a regular language as a finite automaton; *though* it is quite straightforward to check whether a particular string or a particular set of strings belongs to the language *yet* it is quite difficult to infer general or characteristic features of the strings in the language. The problem with the automata approach in this respect lies in the fact that *though* each automaton can be thought of as a mathematical entity in the form of an ordered set of states, set of input symbols etc., *yet* automata are basically **mechanical tools** with the notion of state as primitive. States make generalizations and derivation of formal proofs of the properties of the elements of the languages, extremely difficult. Being basically, mechanical tools, automata are more useful for *implementation* as programs to recognize languages than for *specifying* a language.

For specification of many languages, the **mathematical tool of inductive definitions** has been found quite useful. The inductive definition of the language of natural numbers through Peano's axioms is well-known. An *inductive definition* of a language, that can be defined so, states with some basic concepts, some primitive operations and some axioms that state fundamental properties of the language. The inductive definition of a language captures the *characteristic features* of the defined language and hence facilitates proofs of the language. Also inductive definition facilitates understanding of the language by human beings.

Regular expression is an inductive approach for defining a regular language, and hence, a more useful one for *specifying* a regular language. On the other hand, as mentioned earlier, finite automata approach for defining a regular language is more suitable for implementation as a program to recognize strings of the language. Therefore, in most of the applications involving regular languages, generally the following sequence of four steps, including the three-steps discussed in the previous section, is followed:

- (i) The language of the application under consideration is specified in terms of a regular expression.

- (ii) An equivalent NFA is designed.
- (iii) An equivalent DFA is either designed or simulated.
- (iv) The DFA is then simulated to produce a program that recognizes the strings of the language of the application.

However, for all these applications, built-in routines, *grep*, *regex*, and *lex* for tokenization are available.

The Applications – Introduction

The regular expressions are found useful in the three types of applications:

- (i) For validating inputs,
- (ii) For searching and selecting parts of a given text on the basis of a given pattern,
- (iii) For lexical analysis.

Before, we go into the details of each of these applications, we introduce UNIX type extension of the regular expression notation. We consider below only those extended notations which provide us with more concise notations than normal regular expression notation allows for representing regular languages. Other extensions of regular expression notation, used in UNIX, which may represent non-regular languages are, however, not considered.

Extension of regular expression notation

- (i) the operator '|' is used in place of '+'
- (ii) more than one pair of parentheses may be dropped, if the meaning of the expression does not change. For example, L_{pd} , the language of prime digits may be denoted by
(2|3|5|7)
in stead of the usual r.e. notation
(2 + (3 + (5 + 7)))
- (iii) The symbol '.' (dot) represents 'any character'
- (iv) The notation $[a_1 a_2 \dots a_k]$ represents $a_1 + a_2 + \dots + a_k$. for example, the set of four comparison operators $> < = !$ may be represented by $[> < = !]$ in stead of $> | < | = | !$
- (v) All the characters between x and y , including x and y , in the ASCII sequence may be represented by $[x - y]$. For example, $[A - Z]$ represent all capital letters of the English alphabet. Similarly, the set of all letters and digits may be represented by $[A - Za - Z0 - 9]$.
- (vi) The suffix operator '?' means 'zero or one of'. For example $(xy)?$ stands for $\epsilon | xy$.
- (vii) The suffix operator '+' means 'one or more'. For example, $(xy) +$ stands for $xy | xyxy | \dots$
- (viii) The BNF symbol $:: =$ is used to assign names to languages. For example,
 $\langle \text{digit} \rangle :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$ or
 $\langle \text{digit} \rangle :: = [0 - 9]$

Next, we discuss above-mentioned three types of applications of regular expressions.

Validating inputs from a regular language

One of the common problems with the users of computer system is given incorrect type of inputs. For example, an identifier which is required to have its character as a letter followed by finite number of letters or digits may be incorrectly given by beginning with a dot. Or incomes of individuals in some application are required to be only non-negative integers, but may negative number may by mistake may be given as inputs.

In all such cases, in order to guard against incorrect inputs, programmers use some ad hoc method which is directly programmed into the input routine. A better approach is to use routines such as *regex* that take the input string and a regular expression and then return TRUE, if the input string matches the regular expression specifying the input type. For example, the above mentioned identifiers may be specified by the regular expression.

$$\langle \text{identifier} \rangle ::= [A - Z a - z] ([A - Z a - z 0 - 9])^*$$

And the above mentioned set of non-negative integers may be specified by
 $\langle \text{Non-Neg} \rangle ::= ([0 - 9])^+$

Textual Search & Selection

Next, we explain how regular expressions can be used to locate files and text within files using *grep*¹ Command.

One of the uses of the *grep* command is for searching for patterns specified by regular expressions. The general form of the *grep* command is

$$\text{grep} \langle \text{regular-expression-in-Unix-notation} \rangle \langle \text{filenames} \rangle$$

The utility of the regular expressions in searches can be seen from the way the use of regular expressions simplifies the specification of a problem involving complex searches and then solves the problem using *grep* command. Let us consider the problem of finding all those words in the dictionary word list file say

/user/dict/words

in which all the five vowels occur at least once and one of these occurrences of vowels has the five vowels in lexicographic order, i.e, in one occurrence of the vowels, *a* precedes *e*, *e* precedes *i* then *i* precedes *o* and finally *o* precedes *u*.

It can be easily seen that without the use of regular expressions, the task of specification of the search problem is quite difficult. However the following command

$$\text{Grep} \text{ '.*a.*e.*i.*o.*u' } \text{ /user/dict/words}$$

Easily specifies the problem. The above command prints words like

adventitious
 facetious
 sacrilegious

Let us consider another problem. Suppose we have a directory of a large number of files, where each file is a book on computer science in electronic form. We want to

¹ *grep* stands for "global regular expression print."

know about Turing. However, we do not remember the exact book (s) find the required references through the command

```
grep Turing *
```

Also the technique of specification through regular expressions has been found quite useful in the description of *vaguely defined* class of patterns. As we will illustrate below, because of vagueness, it is not possible to describe the patterns correctly, not only in the first instance, but it may not be possible to describe the pattern correctly after even a number of attempts. However, the regular expression technique allows us gradually better and better specifications of the vaguely defined class of patterns. Next, we illustrate what we have said above through the following example:

Suppose we want to generate a mailing list for the purpose of expansion of our business. Further, we want to focus on recognizing street addresses in particular. But, we do know street parts of the addresses may contain 'Street' or just 'st'. etc. Therefore, to begin with, we use the regular expression

```
Street St\.
```

(Note the backslash is used to override the use of dot, in extended regular expression notation, for any character and not just for the character dot).

Later on, we realize some of the street addresses use to 'Road' or 'Rd.' in stead of 'Street' or 'St.'. Therefore we extend our earlier notation to

```
Street St\ Road Rd\.
```

On further examination, we find even the words 'Puri' or 'Gali' are also used for the purpose.

However, the regular expression technique does not require us to rewrite the whole specification once again. Rather, we extend the earlier specification to

```
Street St\ Road Rd\ Gali Puri
```

Thus the specification for street part of the addresses is improved gradually.

Apart from grep command, regular expressions techniques is an elegant vehicle for specification of arguments for most of the UNIX commands.

Application of Regular Expression in Lexical Analysis

We know that in order to solve problems using computers, most of the solutions of the (solvable) problems are written as programs in some high-level language. But, on the other hand, computers understand and act (directly) only on solutions expressed in machine languages, i.e., expressed as sequences of 0's and 1's. Compilers, alongwith other translators, are computer programs that are used to translate a high-level language program into an equivalent machine language program. The process of translation by compilers called **compilation**, is quite a complex task and, hence is divided into a number of phases. *Well-known phases of compilation are:* Lexical analysis, syntax analysis, Intermediate code generation, Code optimization, Code generation, Table management and Error handling.

Input to lexical analysis phase of compilation are high-level language programs, but program is treated only as a sequence of characters. Sequences of characters do not carry any meaning. The lexical analysis phase converts a sequence of characters of a correctly written program into a sequence of tokens; while for an incorrectly written program, the phase generates an appropriate error message. Tokens in

programming language context, are like *words* in a natural language—they carry meaning. To elaborate further, we may notice that characters like a, b, c, ... etc. and even *string* like *rea* and *run* do not carry any meaning in English language. But *words* like *car* and *run* carry meanings, rather are basic units of meaning of expressions in English. Similarly, *tokens in programming languages are units of meaning of programs*. However, each programming language has its own definition for its legal tokens.

In almost all programming languages, tokens are categorized into different **token classes** viz *keywords, identifiers, literals, operators, separators and comments*. For tokens from different classes, different actions are generated at lexical analysis and later phases. These categories are illustrated as constituents of the following c program fragment

The program computes nth Fibonacci number

```
Void main () {
    int n;
    n = 22;
```

In the above fragment, the first line of the code is a *comment*, the tokens *void* and *int* are *keywords*, the token *n* is an *identifier* (or a name), the token = is an operator; the token 22 is a *literal* and each of the four tokens) ({ and ; is a *separator*.

In this section, we explain how regular expressions are useful in the lexical analysis phase of compilation. In the next section, we discuss how regular expressions are used in syntax analysis phase of compilation.

We have mentioned that a *lexical analyzer* is a module in the overall compiler, a computer program. Lexical analyzer takes sequences of (meaningless) characters as input and returns corresponding sequences of (meaningful) tokens. Writing a lexical analyzer module has been quite a complex task. However, less [1975] suggested a tool, called LEX, for automatically generating lexical analyzers. And, here comes the role for the regular expressions. The **inputs to LEX** is a set of pairs of the form (*regular expression, action for the lexical analyzer for the regular expression*), where each regular expression specifies a token and action is in the form of piece of code which is executed whenever a token specified by the corresponding regular expression, is recognized. The **output of LEX** is a lexical analyzer. Thus LEX, using regular expressions, simplifies the tedious task of developing a lexical analyzer for a given pair of high-level language and a machine level language. It is out of scope for the course to discuss how LEX generates lexical analyzers, or even what actions are associated with different tokens. However, we give below how and what regular expressions are associated with different token classes.

Class	regular expression
digit	[0-9]
alphabet	[a-z A-Z]
integer	[0-9] +
identifier	[a-z A-Z] [a-z A-Z 0-9]*
keyword	"if" "else" "while" "boolean" "int" "real" "man" "void"
literal	[0-9] + "true" "false"
boolean	"true" "false"
whitespace	\f \n \r \r\n \t (invisible space)
comment	"/" [a-z A-Z]* ("r" "\n" "\r\n")

Class	regular expression
separator	} {) (, };
operator	+ - * / < <= > >= = != && !

Next, the above regular-expression based definitions alongwith the codes for appropriate corresponding actions (not defined and discussed here) is fed to the LEX tool, which in turn generates a lexical analyzer. Then any input sequence of characters constituting a program in the high-level language is fed to the lexical analyzer delivered by LEX. If the program is correctly written, then segments of the input sequence are marked as whitespaces, comments or other appropriate token classes. Then, the lexical analyzer takes appropriate action according to the classification of each segment of the input sequence. This, in essence, explains the role of regular expressions in the lexical phase of compilation.

In the next section, we discuss applications of context-free grammars.

3.4 APPLICATION OF CONTEXT-FREE GRAMMARS

In his attempt to describe natural languages, N.Chomsky [1956] conceived the idea of context-free grammar (CFG). Though the attempt has not been completely successful as definitions of natural languages are concerned, yet the idea of CFG has been quite useful in providing definitions of major parts of programming languages. J.W. Backus [1959] and P. Naur [1960] used the idea of CFG in the definition of the syntax of the programming languages FORTRAN and ALGOL respectively. *In fact, several hundred years before christian era, Panini used ideas, somewhat similar to that of context-free grammars, to describe the syntax of Sanskrit language.*

Coming back to Backus and Naur, they in the process of defining respectively FORTRAN and ALGOL, gave an elegant notation, now very well-known as Backus-Naur Form or BNF, to describe programming languages. As BNF is used to describe languages, therefore, the notation of BNF is a meta-language — a language to describe language.

BNF notation can represent any context-free language, i.e., it is equivalent to the vehicle of context-free grammar (CFG) in respect of definitions of languages. However, BNF has the advantage that it uses a small number of symbols, distinct from the symbols used in the programming languages, to define (context-free parts of) programming languages. In order that the notation is easy to read and is more concise, here we use extended BNF which includes some elements of extended regular expressions. The extended BNF notation mainly consists of

- (i) angular brackets, i.e., < > to denote variables e.g, < identifier > denotes the class of identifiers
- (ii) ‘:: =’ is used to denote ‘is defined as’. For example the BNF statement

$$\langle \text{identifier} \rangle :: = \langle \text{letter} \rangle \langle \text{letter} \rangle^* \langle \text{digit} \rangle^*$$
states that an identifier is obtained by writing a letter followed by zero or more of letters and digits
- (iii) the symbol ‘|’, ‘*’ and ‘+’ have the same meanings of ‘or’, ‘zero or more number of times’ and ‘one or more number of times’ respectively as used in the regular expressions.

- (iv) The pair [] of brackets denotes zero or one. For example, [<digit>] denotes zero or one of < digit >

As discussed earlier, CGF is a useful tool for defining programming languages. In this context, we discuss parts of definitions of a C type small hypothetical language and definition of part of HTML. For this purpose, we use BNF notation.

3.4.1 Definition of C-Type Small Language

- (i) Let us first discuss how the concept of CFG, with BNF notation, is used in defining C type of a small hypothetical language.

```

< statement > :: = < assignment statement > | < compound-statement > |
                < selection-statement > | < iteration-statement >

< assignment-statement > :: = < identifier > = < expression >;

< selection - statement > :: = if (< logical-exp >) < statement > |
                              if (< logical-exp >) < statement > else
                              < statement > |
                              switch (< expression >) {<Cases>}

< logical-exp > :: = < comparison > | < comparison > && <logical-exp> |
                  < comparisons > | < logical-exp >

< comparison > :: = (< Bool-operand > < comparison-operator >
                  <Bool- operand >)

< Bool-operand > :: = true | false | < identifier >

< expression > :: = < factor > | < expression > + < factor > |
                  < expression > - < factor >

< factor > :: = < operand > | < factor > * < operand >

< operand > :: = < integer > | < identifier > | (< expression >)

< comparison-operator > :: = > | >= | < | <= | = | !=

< Cases > :: = < Case > < Cases > | < Default >

< Case > :: = < CaseHead > < cases > | < case Head > < statement >

< CaseHead > :: = case < literal > :

< Default > :: = default : < statement-seq >

< iteration-statement > :: = while (< logical-exp >) < statement > |
                              do < statement > while (< expression >) |
                              for ([<expression >]; [< expression >];
                              [<expression>])< statement >

```

Remarks: It may be pointed out here that in almost all programming languages, there are some context-sensitive (i.e., non-context-free) issues. For example, the types of identifiers are declared under declarations. However, in order to check whether an identifier in an expression is of correct type, CFG vehicle is not enough. For this purpose, we use some other non-context-free tool like a symbol-table.

- (ii) In the previous section we discussed how the various token classes are defined specified in terms of regular expressions. However, the token classes, when defined in terms of a CFG notation, their are *more understandable* Below we define some token classes in terms of extended BNF notation.

$\langle \text{input} \rangle ::= \langle \text{whitespace} \rangle \mid \langle \text{comment} \rangle \mid \langle \text{token} \rangle$
 $\langle \text{white space} \rangle ::= \backslash f \mid \backslash n \mid \backslash r \mid \backslash r \backslash n \mid \backslash t \mid \quad (\text{invisible space})$
 $\langle \text{comment} \rangle ::= // \langle \text{sequence of characters} \rangle \backslash r \mid \backslash n \mid \backslash r \backslash n$

The meaning of the above expression in BNF-notation is that a comment in the hypothetical language is denoted by a sequence of characters consisting of

- (a) a pair slashes
- (b) any sequence of characters, terminated by
- (c) either $\backslash r$ or $\backslash n$ or $\backslash r \backslash n$

$\langle \text{token} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{keyword} \rangle \mid \langle \text{literal} \rangle \mid$
 $\quad \langle \text{separator} \rangle \mid \langle \text{operator} \rangle$

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle (\langle \text{letter} \rangle \mid \langle \text{digit} \rangle)^*$

$\langle \text{letter} \rangle ::= a \mid b \mid c \mid \dots \mid y \mid z \mid A \mid B \mid \dots \mid Y \mid Z$

$\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid \dots \mid 8 \mid 9$

$\langle \text{keyword} \rangle ::= \text{if} \mid \text{else} \mid \text{while} \mid \text{boolean} \mid \text{int} \mid \text{real} \mid \text{main} \mid \text{void}$

$\langle \text{literal} \rangle ::= \langle \text{integer-value} \rangle \mid \langle \text{boolean value} \rangle \mid \langle \text{real-value} \rangle$

$\langle \text{integer-value} \rangle ::= (\text{digit})^+$

$\langle \text{Boolean-value} \rangle ::= \text{true} \mid \text{false}$

$\langle \text{real-value} \rangle ::= \langle \text{decimal} \rangle \mid \langle \text{exponential} \rangle$

$\langle \text{decimal} \rangle ::= \langle \text{signed-integer} \rangle . \langle \text{integer} \rangle \mid \langle \text{signed-integer} \rangle . \langle \text{sign} \rangle .$
 $\quad \langle \text{integer-value} \rangle$

3.4.2 Definition of Part of HTML

Next, we define a part of HTML. In the following, we use ' \rightarrow ' in stead of ' $::=$ ' and we use the underscore in stead angular brackets, e.g., in stead of

$\langle \text{type} \rangle$ we use type

HTML-Document $\rightarrow \langle \text{html} \rangle \underline{\text{document}} \langle / \text{html} \rangle$

Document $\rightarrow \langle \text{head} \rangle \underline{\text{head-part}} \langle / \text{head} \rangle \langle \text{body} [\text{body-attributes}]^* \rangle$
 $\quad \underline{\text{body-part}} \langle / \text{body} \rangle$

head-part $\rightarrow [\langle \text{title} \rangle \underline{\text{title-part}} \langle / \text{title} \rangle]$

title-part $\rightarrow \underline{\text{string}}$

string $\rightarrow [\underline{\text{letter}}] [\underline{\text{letter}} \mid \underline{\text{digit}}]^*$

digit → 0 | 1 | 2 | ... | 8 | 9

letter → a | b | ... | y | z | A | B | ... | Y | Z

body-attributes → [background = "background-value"] [bgcolor = "color"]
[text = "color"]

Color → aqua | black | blue | fuchsia | gray | green | lime | maroon | navy | olive | purple |
red | silver | teal | yellow | white | ...

Remarks:

- (i) *The set of available colors may change and is generally enhanced from time to time*
- (ii) *The background-value can not be specified here. However, we may specify some website address for sources of background, e.g.,
< a href = "http://www.gamini.com" > background-value *

body-part → comment | paragraph | textual | line-break | image | linking | map-code | form-code

Comment → < ! - body-part -- >

Paragraph → < p [para - attribute] * body-part < /p >

Para-attribute → align = "align-attribute"

align-attribute → left | right | center

textual → [text] body-part

text → string | [< text-marker > text < / text-marker >] *

text-marker → header-marker | physical-text-marker | content-based-marker |
list-marker

header-marker → h1 | h2 | h3 | h4 | h5 | h6

physical-text-marker → b | big | blink | i | s | small | sub | sup | tt | u

content-based-marker → cite | code | dfn | em | kbd | samp | strong | var

list-marker → unordered-list-marker | ordered-list-marker | definition-list-marker

unordered-list-marker → < ul > listed-items < /ul >

listed-item → < b_i > string < /b_i >

ordered-list-marker → < ol > listed-items < /ol >

definition-list-marker → < dl > definition-list < /dl >

definition -list → [defintion] *

definition → < dt > string < /dt > < dd > string < /dd >

line-break → < br > body-part

image → < img src = "path" [blank image-attribute]* > blank → nbspnbsp;
image-attribute → alt = "string" | align = "align-attribute" |
integer-attribute = "integer"
align-attribute → left | right | top | texttop | middle | absmiddle | centre | bottom |
baseline | absbottom
integer-attribute → height | width | hspace | vspace | border | integer → [digit]*
linking → < a href = "path" > linked-item-identifier
linked-item-identifier → string |
[image] + path → absolute-path | relative-path
absolute-path → [web-path | non-web-path] relative-path
web-path → web-protocol DNS-name of host
web protocol → web-protocol-name:
web-protocol-name → http | ftp | gopher | telnet
DNS-name-of-host → // [www | ftp] [. String] + [: port]
port → integer
non-web-path → non-web-protocol mail-address
non-web-protocol → news | mailto
mail-address → string @ string [.string]*
relative-path → / directory/file

3.5 SUMMARY

In this unit, we discussed applications of some of the models of computation developed in the earlier units of the course:

- first of all, we explained the process of the design of finite-automata based algorithms for search of text on the Web or of any on-line textual database.
- next, we explained how regular expressions are used for (i) validating inputs, for searching and selecting parts of a given text on the basis of a given pattern (ii) for lexical analysis.
- finally, we illustrated how Backus-Naur Form (BNF) notation can be used for defining (the syntax of) context-free parts of programming languages through the examples of definition of a small C-type language and definition of a subset of HTML.

3.6 FURTHER READINGS & REFERENCES

- (i) John E. Savage, Alan L. Selman, and Carl Smith [2001]: "History and Contributions of Theoretical Computer Science" in *Advances in Computers* Vol. 55.
- (ii) M.E. Lesk [1975]: "LEX — a lexical analyzer generator", *CSTR 39, Bell Laboratories, Murray Hill, N.J.*
- (iii) N. Chomsky [1956]: "Three models for the description of language," *IRE Trans. on Information Theory* 2:3, pp. 113-124.
- (iv) J.W. Backus [1959]: "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference" *Proc Intl. Conf. On Information Processing (1959)* UNESCO, pp. 125-132.
- (v) P. Naur et al. [1960]: "Report on algorithmic language ALGOL 60," *Comm. ACM* 3:5 (1960) pp. 299-314. See also *Comm. ACM* 6:1 (1963), pp. 1-17.
- (vi) J.E. Hopcroft, R.Motwani & J.D.Ullman [2001]: Introduction to Automata Theory, Languages, and Computation (II Ed.) *Pearson Education Asia*.

EPILOGUE: FOR THOSE WHO CARE....

Applications

... specially those from amongst us – the teachers, particularly of computer science, from institutions of higher learning, and, of course, the students:

I believe these three topics – ethics, professional behaviour, and social responsibility – must be incorporated into the computer science curriculum. Personally I do not believe that a separate course on these topics will be effective. From what little I understand of the matter of teaching these kinds of things, they can best be taught by example, by the behaviour of the professor. They are taught in the odd moments, by the way the professor phrases his remarks and handles himself. Thus it is the professor who must first be made conscious that a significant part of his teaching role is in communicating these delicate, elusive matters and that he is not justified in saying, "They are none of my business." These are things that must be taught constantly, all the time, by everyone, or they will not be taught at all. And if they are not somehow taught to the majority of our students, then the field will justly keep its present reputation (which may well surprise you if you ask your colleagues in other departments for their frank opinions).

R.W. Hamming
in
Turing Award Lecture (1968)

R.W. Hamming emphasizes below the need for treating computer science as more of an engineering discipline than as pure mathematics.

...For example, let me make an arbitrary distinction between science and engineering by saying that science is concerned with what is possible while engineering is concerned with choosing, from among the many possible ways, one that meets a number of often poorly stated economic and practical objectives. We call the field "computer science" but I believe that it would be more accurately labeled "computer engineering" ...

...I would like to see far more of a practical, engineering flavor in what we teach than I usually find in course outlines...

...At the heart of computer science lies a technological device, the computing machine. Without the machine almost all of what we do would become idle speculation, hardly different from that of the notorious Scholastics of the Middle Ages...

...I still believe that it is important for us to recognize that the computer, the information processing machine, is the foundation of our field. How shall we produce this flavor of practicality that I am asking for, as well as the reputation for delivering what society needs at the time it is needed? ...

...We need to avoid making computer science look like pure mathematics: our primary standard for acceptance should be experience in the real world, not aesthetics...

...Without real experience in using the computer to get useful results the computer science major is apt to know all about the marvelous tool except how to use it. Such a person is a mere technician, skilled in manipulating the tool but with little sense of how and when to use it for its basic purposes. I believe we should avoid turning out more idiot savants – we have more than enough "computniks" now to last us a long time. What we need are professionals! ...

R.W. Hamming
in
Turing Award Lecture (1968)

SUMMARY OF

ACM Code of Ethics and Professional Conduct

Adopted by ACM Council 16 Oct. 92.

- Preamble
- Contents & Guidelines

Preamble

Commitment to ethical professional conduct is expected of every member (voting members, associate members, and student members) of the Association for Computing Machinery (ACM).

This Code, consisting of 24 imperatives formulated as statements of personal responsibility, identifies the elements of such a commitment. It contains many, but not all, issues professionals are likely to face. Section 1 outlines fundamental ethical considerations, while Section 2 addresses additional, more specific considerations of professional conduct. Statements in Section 3 pertain more specifically to individuals who have a leadership role, whether in the workplace or in a volunteer capacity such as with organizations like ACM. Principles involving compliance with this Code are given in Section 4.

The Code shall be supplemented by a set of Guidelines, which provide explanation to assist members in dealing with the various issues contained in the Code. It is expected that the Guidelines will be changed more frequently than the Code.

The Code and its supplemented Guidelines are intended to serve as a basis for ethical decision making in the conduct of professional work. Secondly, they may serve as a basis for judging the merit of a formal complaint pertaining to violation of professional ethical standards.

It should be noted that although computing is not mentioned in the imperatives of Section 1, the Code is concerned with how these fundamental imperatives apply to one's conduct as a computing professional. These imperatives are expressed in a general form to emphasize that ethical principles which apply to computer ethics are derived from more general ethical principles.

It is understood that some words and phrases in a code of ethics are subject to varying interpretations, and that any ethical principle may conflict with other ethical principles in specific situations. Questions related to ethical conflicts can best be answered by thoughtful consideration of fundamental principles, rather than reliance on detailed regulations.

Contents & Guidelines

1. General Moral Imperatives.
2. More Specific Professional Responsibilities.
3. Organizational Leadership Imperatives.
4. Compliance with the Code.
5. Acknowledgments.

1. GENERAL MORAL IMPERATIVES.

As an ACM member I will

- 1.1 **Contribute to society and human well-being.**
- 1.2 **Avoid harm to others.**
- 1.3 **Be honest and trustworthy.**
- 1.4 **Be fair and take action not to discriminate.**
- 1.5 **Honor property rights including copyrights and patent.**
- 1.6 **Give proper credit for intellectual property.**
- 1.7 **Respect the privacy of others.**
- 1.8 **Honor confidentiality.**

2. MORE SPECIFIC PROFESSIONAL RESPONSIBILITIES.

As an ACM computing professional I will

- 2.1 **Strive to achieve the highest quality, effectiveness and dignity in both the process and products of professional work.**
- 2.2 **Acquire and maintain professional competence.**
- 2.3 **Know and respect existing laws pertaining to professional work.**
- 2.4 **Accept and provide appropriate professional review.**
- 2.5 **Give comprehensive and thorough evaluations of computer systems and their impacts, including analysis of possible risks.**
- 2.6 **Honor contracts, agreements, and assigned responsibilities.**
- 2.7 **Improve public understanding of computing and its consequences.**
- 2.8 **Access computing and communication resources only when authorized to do so.**

3. ORGANIZATIONAL LEADERSHIP IMPERATIVES.

As an ACM member and an organizational leader, I will

- 3.1 **Articulate social responsibilities of members of an organizational unit and encourage full acceptance of those responsibilities.**
- 3.2 **Manage personnel and resources to design and build information systems that enhance the quality of working life.**

- 3.3 Acknowledge and support proper and authorized uses of an organization's computing and communication resources.
- 3.4 Ensure that users and those who will be affected by a system have their needs clearly articulated during the assessment and design of requirements; later the system must be validated to meet requirements.
- 3.5 Articulate and support policies that protect the dignity of users and others affected by a computing system.
- 3.6 Create opportunities for members of the organization to learn the principles and limitations of computer systems.

4. COMPLIANCE WITH THE CODE.

As an ACM member I will

- 4.1 Uphold and promote the principles of this Code.
- 4.2 Treat violations of this code as inconsistent with membership in the ACM.

Let us strive to comply with the above-mentioned moral and leadership imperative and follow the code of professional responsibilities.