BIJU PATNAIK UNIVERSITY OF TECHNOLOGY, ODISHA

Lecture Notes

On

THEORY OF COMPUTATION

MODULE - 2

UNIT - 2

Prepared by,

Dr. Subhendu Kumar Rath,

BPUT, Odisha.

UNIT 2 TURING MACHINE – MISCELLANY

Structure **Page Nos.**

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 $\{x^n y^n : n \in N\}$ 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.***
Next, we discuss ***Universal Turing Machine (UTM)***, ***an equivalent of*** *general-purpose* ***computer***. **The significance of the study of UTM lies in the facts**:

. . . .

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)

(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 configrations* 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 configration.**

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 configrations 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 # #..#
          ↑
          q2
```

then the configuration in the *standard* TM is written as:
$(q_2,\ \#\ \#\ a\ b\ \#\ c\ \underline{d}\ e\ f)$,
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

```
...##...#....##a b # c d  e f #...#...
                    ↑
                    q2
```

and then, the configuration for Two-way infinite tape TM will be slightly different as given below:

(q_2, a b # c <u>d</u> e f),
Note the #'s to the left of a are missing here.

(ii) ***No Hanging (or No ceasing of operations without Halting)***
In this case, as there is no left end of the tape, therefore, there is no possibility of jumping off the left-end of the Tape. Thus, if the machine has the configuration (q, <u>a</u> d ...) and δ (q, a) = (p, b, L), *then new configuration is (p, <u>#</u> b d ...) instead of the hanging configuration.*

(iii) ***The empty Tape configuration:*** When at some point of time all the cells of the Tape are #'s and the state is say q, then the configuration *in Two-way Tape* may be denoted as:
(q, <u>#</u>)

where only the current cell containing # is shown in the configuration.
Rest of the notations and definitions given in context of standard TM will be used for two-way Turing Machine, including the definition of the next-move (partial) function δ.

Despite the fact that, it is possible in the new model of computer to move left as far as required; as mentioned earlier, the model does not provide any additional computational capability.

2.2.2 Extension (ii):

Turing Machine having R heads, k ≥ 2, with only oneTape

In order to simplify the discussion, *we assume that there are only two Heads on the Tape.*

The Tape is assumed to be one-way infinite. We explain the involved concepts with the help of an example.

Let the contents of the Tape and the position of the two Heads, viz H_1 and H_2, *be as given below:*

$$\#\#a\ \underset{\uparrow}{b}\ c\#\ d\ \underset{\uparrow}{e}\ f\#\#.... \qquad \text{..... (*)}$$
H_2 H_1

Further, let the state of the TM be q.

Then one method of defining **the configuration of two-Head one-way Turing machine** is
*(the state, the Tape description as if H_1 is the **only** Head of TM, the Tape description as if H_2 is the **only** Head of TM).*

Therefore, the configuration in the case of (*) given above will be

(q, # # a b c # d <u>e</u> f, # # a <u>b</u> c # d e f)

The **Move function of the Two-Head One-way Turing Machine may be defined** as
δ (state, symbol under Head 1, Symbol under Head 2)

$$= (\text{New State}, (S_1, M_1), (S_2, M_2))$$

Where S_i is the symbol *to be written* in the cell under H_i, the ith 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
δ (q, symbol under H_1, 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:

Tape 1:

#	#	a	b	c	d	e	#	#	
			↑						

Tape 2:

e	f	#	g	d	f	#	#	#	….
				↑					

and *the state* of the Turing Machine *be q*.
Then the ***configuration*** may be denoted by
(q, (# # a <u>b</u> c d e), (e f # g <u>d</u> f)))
(*inner pairs of parentheses are used only to enhance readability, not required otherwise*)
The next Move function δ may be defined as

δ ((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 ith 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 ith Tape in place of T_i. $M_i \in \{L, R, N\}$ denotes the movement of the Head on ith 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 l and M_1 being 'Movement to Left', or
(ii) *T_2 being is 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.

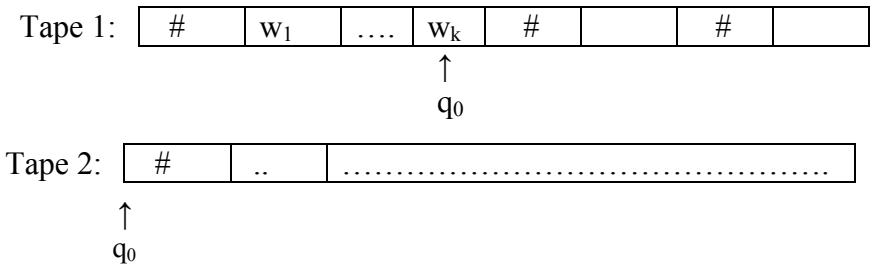
A very important application of the 3-tape Turing Machine model, which we are going to discuss in Section 2.3, is in the design of universal Turing Machine, a sort of a general-purpose computer.

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 # ω ω $\underline{\#}$ for given input # ω $\underline{\#}$.

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 configration is as follows:



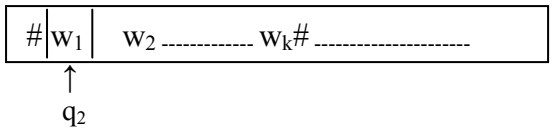
Step1: Move the Head of Tape 1 containing the input towards the left most cell through the following moves.

$\delta (q_o, \#, \#) = (q_1, (\#,L), (\#, N))$
$\delta (q_1, \bar{\#}, \#) = (q_1, (\bar{\#},L), (\#, N))$
$\delta (q_1, \#, \#) = (q_2, (\#,R), (\#, R))$

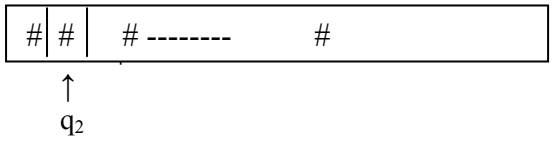
where $\bar{\#}$ denotes the same non-blank symbol throughout an equation.

After these moves, the configuration is as follows:

Tape 1:



Tape 2:



where $w_i \neq$ # for i=1,2,-------, k

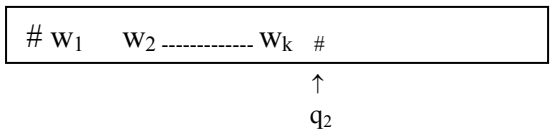
Step 2: Next, we copy the contents of Tape 1 to Tape 2 through

$\delta (q_2, \overline{\#}, \#) = (q_2, (\overline{\#}, R), (\overline{\#}, R))$,

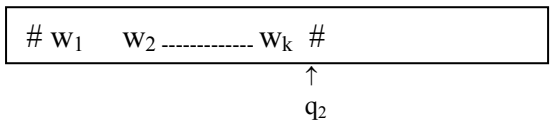
where $\overline{\#}$ *denotes the same non-blank symbol throughout an equation.*
In other words through these k moves, non-blank contents of Tape 1 are copied in the corresponding cells of tape 2.

After k times executions of the above move, the configuration becomes

Tape 1:



Tape 2:



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:

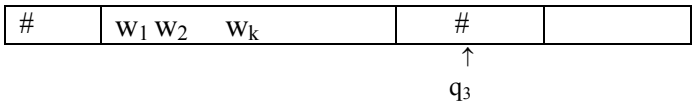
$\delta (q_2, \#, \#) = \delta (q_3, (\#, N), (\#, L))$

and

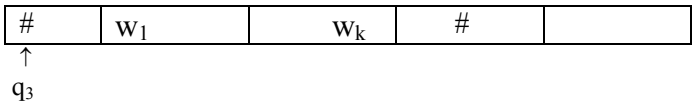
$\delta (q_3, \#, \overline{\#}) = (q_3, (\#, N) (\overline{\#}, L))$

At the end of k moves the configration becomes

Tape 1:



Tape 2:



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

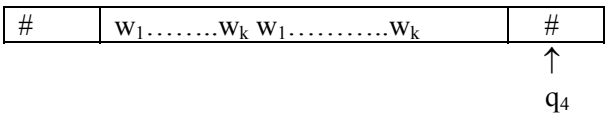
$\delta (q_3, \#, \#) = (q_4, (\#, N), (\#, R))$ **....(*)**

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,

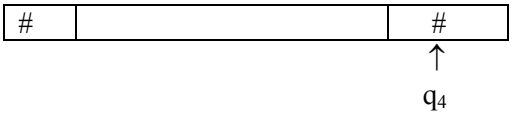
$\delta(q_4, \#, \overline{\#}) = (q_4, (\overline{\#}, R), (\#, R))$

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 $w = w_1w_2\text{------}w_{k-1}\ w_k$
then $w^R = w_k\ w_{k-1}\text{--------}w_2\ w_1$

Remark 2.2.3.2:

Again, it has been proved that the power of the *standard Turning* Machine is the same as that of a *Turing Machine with any finite number of Tapes*.

Remark 2.2.3.3:

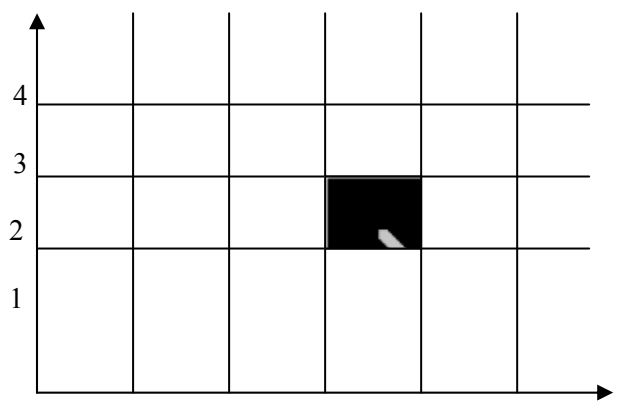
The *k-Tape version of a Turing Machine*, with each tape being only one-way can be further extended to a k Tape Turing Machine with each *Tape being Two way infinite*. It may again be noted that *even with this extension* the computing power *is the same* as is achievable with standard TM.
Next, let 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*.



0

0 1 2 3

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 $q \in Q$, $c_k \in \Gamma \sim \{\#\}$.

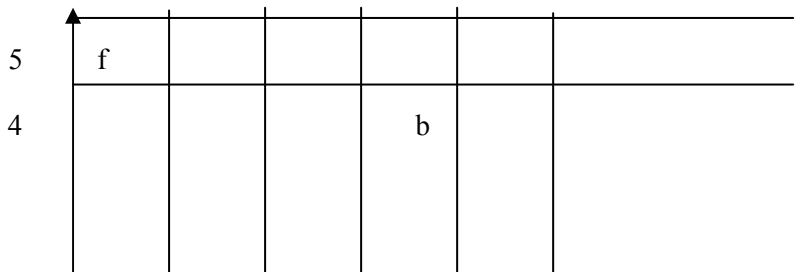
i.e. c_k is a non-blank Tape symbol.

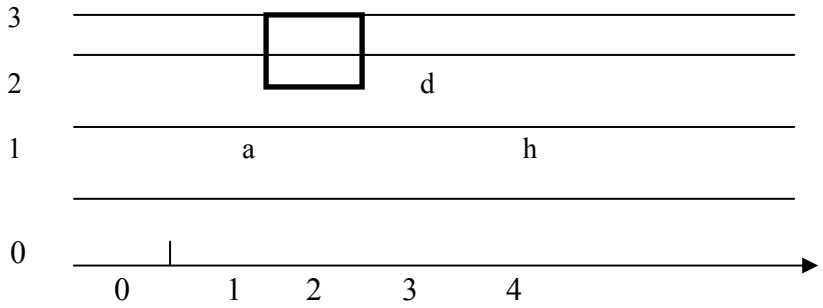
Then a configuration at a particular instant is denoted by
$(q, (H_1, H_2) (i_1, i_2, c_{i1, i2}), (j_1,j_2,c_{j1, j2}), \ldots..,\ (k_1, k_2, c_{k1, k2}) \ldots\ldots\ldots\ldots)$),
where each of $c_{i1,i2}$, $c_{j1,j2}$, …….. is non-blank and these are the only non-blanks on the tape.

Also, (H1,H2) 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*
$i_1 \le j_1 \le k_1$.
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 x Γ **to Qx** Γ x { 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,**
δ **(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 δ (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 ≠ # then (i_1, i_2, d) will be added to the set of triplets in the configuration.

Case (ii) if c ≠ # 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
δ (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
δ (q_2 c) = (d, n).

Case (iv) When c ≠ # ,and d ≠ #, 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
δ (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,…….} 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 the highlighted. By the definition of δ which maps an element of (q, x) of Q x Γ to a set $\{(q_i, x_i, M_i)\}$ **means that each element (q, x) of Q x Γ 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 x Γ to Q x Γ x***{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 x Γ x {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 are 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 $(Q, \Sigma, \Gamma, \delta, q_o, h)$ where
Q: Set of States

Σ: Set of input symbols
Γ : Set of tape symbols
q_o: The initial state
h: The halt state and

δ: Q x Γ → **Power set of** (Q x Γ x {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 staring 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
$\alpha = a_1 a_2 \ldots a_k \in$ L for $a_{ii} \in \Sigma$, the set of input symbols of M, then
$(q_0, a_1 a_2 \ldots a_k) \vdash^* (h, \beta)$

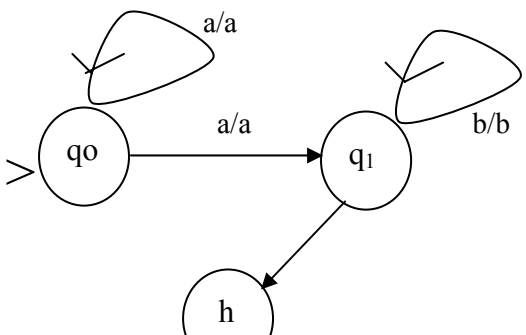
Where β is a string of tape symbol and tape head may be on any cell of the tape. A characteristic feature of the **standard TM**, in this case, is that if there is to be a sequence of moves from (q_0, α) to a final state, than that sequence might the unique. However **in the case of Non-Deterministic 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 ≥ 1, m ≥ 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:



b/b

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, #}, δ, q_o, h }
Where δ is defined as follows:
δ (q_0, a)= {(q_0, a, R), (q_1, a, R)}
δ (q_0, b)= empty
δ (q_1, a)= empty
δ (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 pursume 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, Σ, ⌈, δ, q_0, F)

where the various involved symbols denote various entities as follows:

Q	:	The set of states
Σ	:	The set of input symbols
⌈	:	The set of Tape symbols
q_0	:	The initial state
F	:	The set of finial states and
δ	:	is a partial function *from* Qx⌈ *to* Q x⌈ x { L, R, N }, with L, R and N respectively denoting *move to the Left, move to the Right* and *No move of the Head*

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
$\{a^n b^m : n \geq 1, \quad m \geq 0\}$

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.*
$\{(q_i, a_j, q_k, a_l, m_f) : q_i, q_j \in Q_M; a_j a_l, \in \Gamma_M; m_F \in \{$ 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,
$Q_\infty = \{q_0, q_1, q_2 \ldots\ldots\}$
and the set of Tape Symbols is
$\Gamma_\infty = \{a_1, a_2, a_3 \ldots\ldots\ldots\}$
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 θ_∞, Γ_∞ and H_{mv} in terms of the alphabet{0,1}, as follows:**
$\lambda(q_i) = 0^{i+1}$ i = 0, 1, 2,……..
(for example $\lambda(q_0) = 0$, $\lambda(q_3) = 0000$, to be denoted by 0^4 etc)
λ $(a_j) = 0^j$ for j = 1, 2, 3…….
(for example $\lambda(a_2) = 00$, to be denoted by 0^2; $\lambda(a_4) = 0000$, to be denoted by 0^4
Also, λ (L)= 0, λ (R)= 00, *(or 0^2)* and λ (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 $(q_i, a_j, q_k, a_l, m_f)$ the components q_k, a_l and m_f *are uniquely determined by* the pair of q_i and a_j and hence **we use the shorthand M_{ij} for the move $(q_i, a_j, q_k, a_l, m_f)$.**

By the above-mentioned coding scheme, the five components q_i, a_j, q_k, a_l and m_f are respectively represented as $\lambda(q_i)$, $\lambda(a_j)$, $\lambda(q_k)$, $\lambda(a_l)$ and $\lambda(m_f)$, each of which is a sequence of 0's.

Next the move M_{ij} given by $(q_i, a_j, q_k, a_l, m_f)$ 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, the 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 three1'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_o and λ the Next-Move function.*

In view of these notational conventions, the code of a TM, may be given by

$$111\ \lambda(q_o)1\ \lambda(M_{11})1\ \lambda(M_{12})1\ \lambda(M_{14})\ldots 1\ \lambda(M_{21})1\ \lambda(M_{22})\ \ldots\ldots\ 1\ \lambda(M_{mn})11$$

…….. **(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
$\Gamma_\infty = \{a_1, a_2\ldots\ldots\}$
denotes the set of countably infinite tape symbols and each of the tape symbols a_j, will be coded as

$\lambda(a_j) = 0^j$ for j = 1, 2, 3 ..…….. **(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 $10^2\ 10^4\ 1\ 0^7\ 1$.

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 I'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_j 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_j). which uniquely determines the move $M_{ij} = (q_i,a_j,a_k,a_l,m_f)$. The control unit extracts the quintuple $(q_i, a_j, q_k, a_l, m_f)$. 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_f 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_f. 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 $a, b, c \in R$, 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

$a \neq 0, b, c \in R$ is equivalently represented by the infinite set $\{(a, b, c), a, b, c \in R$ 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) were 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,.., 9}. And hence each triplet can be written as a sequence over the alphabet {0 1 ….. 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,≠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 Y. However, if $\omega \notin L$, then M **may not halt** further if the Turing Machine halts, on an input ω with $\notin L$ then it should halt with an output different from 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 ∑, is said to be Turing Decidable, if there is a Turing Machine M which always halts when given any input $w \in \Sigma^*$ whether $\omega \in L$ or $\omega \notin L$. Further if $\omega \in L$ then M halts with output Y, indicating that the string ω is in the language L. And if $\omega \notin L$, then M halts with output N, indicating that ω 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 ω, where $\omega \notin L$, but there must be *at least one* string $\omega \notin L$ 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 ω 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.

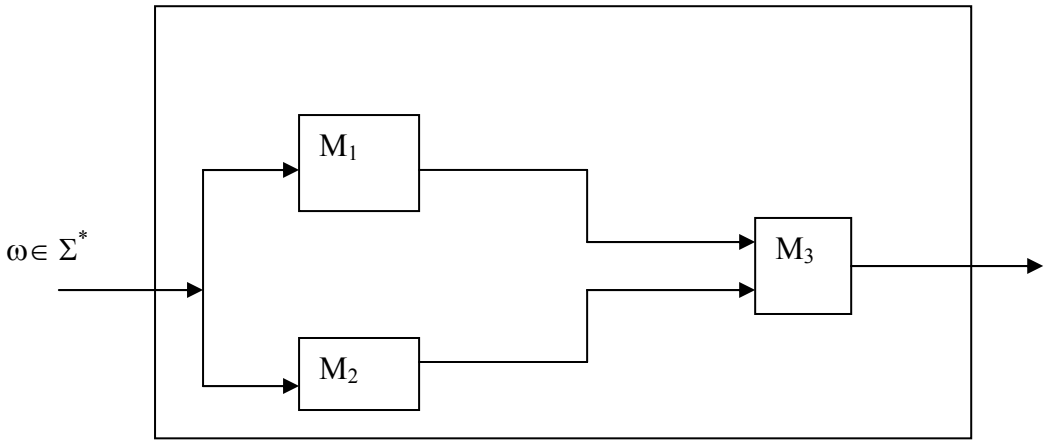
As languages are sets (of strings), therefore, we can talk of union, intersection, and complementation etc. of languages.

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 Y else returns N .
For establishing Part (i) above: we first of all, construct a newTuring Machine M_3 having {Y, 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 Y as output, if at least one of the outputs of M_1 or of M_2 is a Y,

However, if there is no Y in the input to M_3 then the machine returns 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.

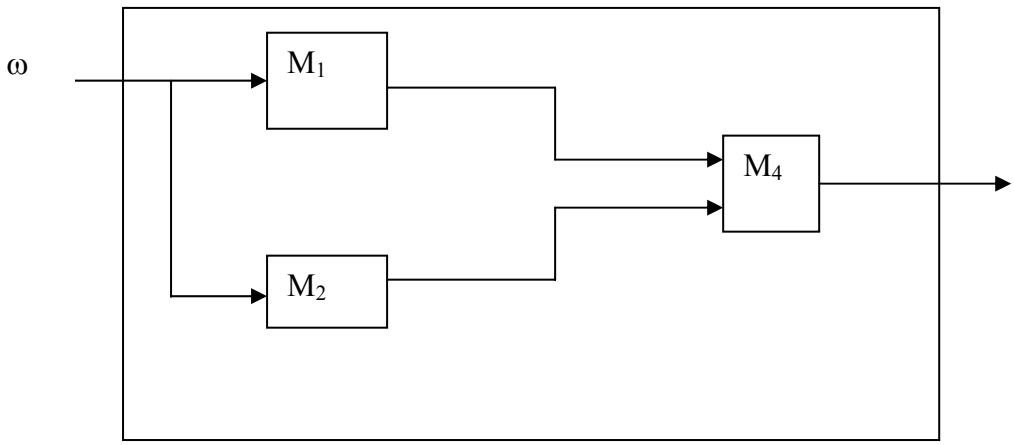


Next, we briefly explain the functioning of the designed machine M-union A string $w \in \Sigma^*$, when given as input to M-union, is further given by the control of M-union, as inputs to both M_1 and M_2.

As both languages are decidable, therefore, after some finite amount of time, both halt, each with an output as Y or 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 Y *if* $\omega \in L_1$, or $\omega \in L_2$, *else* the machine halts with output N. The output of M_3 is the output of M-union.

Thus, for each $w \in \Sigma^*$, M-union returns a Y or N and hence its language $L_1 \cup L_2$ is Turing Decidable.

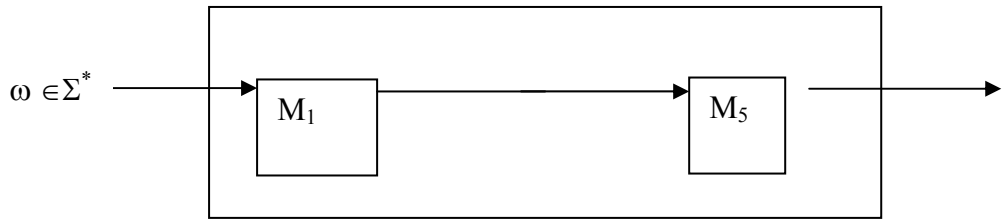
Part (ii) In this case, first of all, we construct a TM M4 having {Y N} as set of input symbols. These input symbols, as mentioned earlier, are the only possible outputs of each of M_1 and M_2. These outputs whenever available are written on the Tape of M_4 as inputs to M_4. The machine M_4 is designed such that it returns a Y if the input sequence consists of both Y's. However, if the input sequence consists of at least one N then M_4 returns 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 Y if both M_1 and M_2 return a Y, else M_4 returns N. And the output of m_4 is the output of M-unit. Thus for each $\omega \in \Sigma^*$, returns either a Y or N in such manner that if $\omega \in L_1 \cap L2$ then M-intersection returns a Y as output, else 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 Y returns N and on reading an N returns a Y .

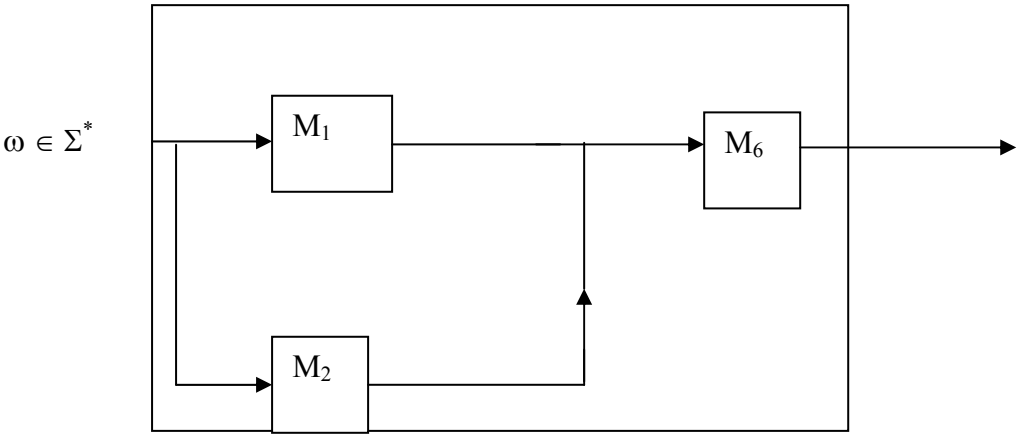
The required TM machine M-complement the following diagrammatic representation.



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 as decides the language L_1, therefore, for $\omega \in L_1$ after a finite number of moves, M_1 outputs Y which is then given as input to M_5, which in turn returns N. Similarly, for $\omega \notin L_i$, M_5 returns 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 Y or N s.t, if $\omega \in L_1$ then M-complement returns N, else returns Y. Hence the language of M-complement is Turning-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 $\omega \notin L_i$, but may or may not halt if $\omega \notin L_i$.
Then a TM M-A-union with the following configuration and description accepts $L_1 \cup L_2$.

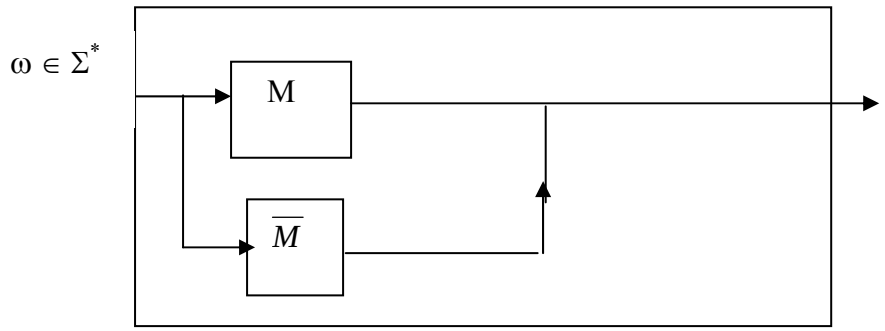


The overall control in M-A-Union which may stop and start any or all of M_1, M_2 and M_6. **The TM M_6 functions as follows: If** , at any stage, there is an output from any one of M_1 or M_2, then on the first output from either M_1 or M_2, the machine M_6 is activated and the output from M_1 or M_2, whichever is available, is written on the tape of M_6. If the output is Y either from M_1 or M_2, say M_1, then the control of the overall machine M returns a Y and halts the machine. **However**, if it is an 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 Y, then overall machine M gives the output Y and Halts. If M_2 halts with an N , then 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 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 $\overline{L} = \Sigma \sim L$ are Turing Acceptable, then L is Turing Decidable.

Proof: Let M and $\overline{M}$ be the TMs that accept respectively the languages L and $\overline{L}$. 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.



UM

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 $\overline{M}$ and activates both M and $\overline{M}$. Whenever an output *if at all*, comes out of M or $\overline{M}$ then the overall machine UM gives output and Halts.

After following actions: If w ∈ L then M halts with output Y. In this case the overall control returns Y as the output of UM. Further, if case $\omega \notin L$ then $\overline{M}$ halts and returns Y. The overall control on checking a Y from $\overline{M}$ returns N as output of UM. Thus for $\omega \in \Sigma^*$, the machine UM always halts and returns Y if $\omega \in L$ and returns 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 $a x^{10} + bx^5 + c = 0$, 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 $L = \{ a^n b^n c^n \geq 0\}$
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 <M, w>,where α_1 is a binary representation of a TM M and α_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… according to lexicographic listing of their binary representations. Similarly, all input strings can also be listed as w_1, w_2, w_3.... 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, ……, u_k, ……..}
where string u_k is obtained from the row labeled T_k in the above table, by inverting its kth bit and keeping all other bits unchanged. For example, u_1 = 0 001 ……. Which is obtained from the row labeled as T1 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

Remark2.5.2.1:

***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_i and α_2, some binary word, such that M_i 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 is not acceptable by the TM having w as its binary code}
= { w : w is not a representation of any TM} $\cup$ ***{ w : w is binary code of a TM say M_j but M_j does not accept w}***

L_u = { < M, w> : < M, w> is binary code representing the pair (M, w) where M is a Turing Machine that accepts w}

Therefore

$\overline{L_u}$ ***= { α : α is a binary string s.t either $\alpha \neq$ < M, w> or if α = < M, w> then M does not accept w}***

= { α : α does not represent ordered pair of a TM and an input string} $\cup$ ***{ α : α = < M, w> and M does not accept w}***

Remark 2.5.4.2:

We may note there is parallel between each pair of languages L_d and $\overline{L_u}$ and the languages $\overline{L_d}$ and L_u. 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_d and $\overline{L_u}$ is that a member of L_d ***represents only inputs w to TMs whereas the each member of L_u is a binary string the form <M, w> 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_u***

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 kth 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

Theorem2.5.5.1:

The language L_u of all binary strings α 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 <u>M</u> that accepts L. We only sketch below the design of the required TM <u>M</u>, 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 Tape1
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 <u>M</u>. Initially, q_0, the initial state, is written of Tape 3.

Step 2: The Process of Simulation of M by <u>M</u>

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 <u>M</u> also knows the state q of the simulated machine M from Tape 3. From the known pair (q, v) the control of <u>M</u> finds from Tape 1 the value (p, u, m) s.t δ_M (q, v) = (p, u, m)' where δ_M is the next-move function of M. At this stage, the control of <u>M</u> 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.

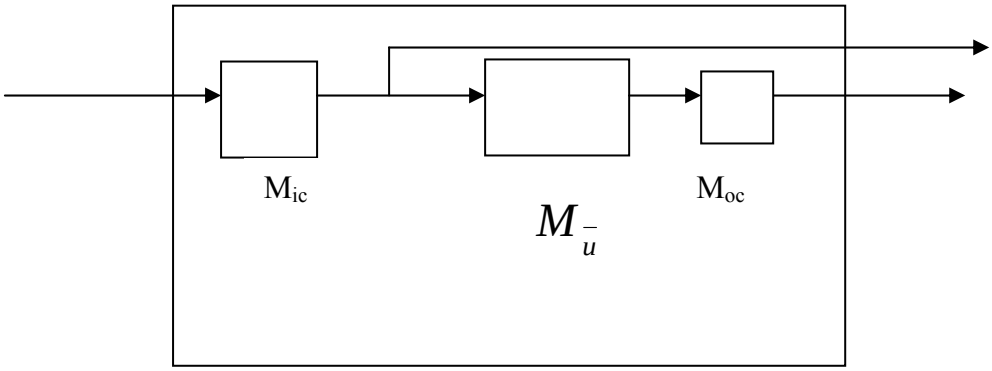
If M accepts w, then the whole process is repeated till we reach halt state of M in which case the control of M returns **'yes'** and if required, waits for the next (M, w) pair to be written on Tape 1 and whole process is repeated.

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_d. But we know L_d 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 $M_{\bar{u}}$ that accepts the language $\overline{L_u}$. We intend to design a TM M_d which accepts L_d using $M_{\bar{u}}$ 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 $M_{\bar{u}}$. In other words, M_{ic} converts a member of L_d into a member of $\overline{L_u}$
(ii) $M_{\bar{u}}$, being a Deciding machine for $L_{\bar{u}}$ 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 $M_{\bar{u}}$ is assumed to be already designed TM that decides language $L_{\bar{u}}$, 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 $L_{\bar{u}}$ represent equivalent languages except the form of its members. Therefore, responses of machines M_d and $M_{\bar{u}}$ 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 $\bar{M}_u$.

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 $\bar{M}_u$. This completes the construction of M_{ic} and hence of M_d which decides L_d if $\bar{M}_u$ decides $\bar{L}_u$.

But, as L_d is Turing undecidable, there can not be a TM M_d deciding it. Hence no $\bar{M}_u$ deciding $\bar{L}_u$ can exist leading to the conclusion that $\bar{L}_u$ is undecidable.

Remark 2.5.5.1:

The proof given above in support of the truth of the statement 'L_u 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 grammer is in the *formal* sense, we recall the definition of a context-free grammer. (In the literature, there are many variations *of the following definition).*

A context-free grammer 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 grammer 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 grammers. 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

*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 Grammers, *Information and control 2:* 137- 167,1959

formal grammer/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 Types 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-there and even as unrestricted languages.

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 $\sum$ each of which a TM T can be designed which halts for every string $\alpha \in \sum^*$, 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.

2.6.1 Grammers 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 grammer. 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 grammer.

Right Linear Grammer: A context-free grammer
G = (V,T,P,S)
is said to be right linear if every production in P is of the form
$A \rightarrow a$ or $A \rightarrow a\,B$,
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 Grammer: A context-free grammer

---

* refer PP. 327-330 Introduction to Languages and the Theory of computation by John C. Martin (TMH, 1998)

G = (V, T, P, S)
is said to be left linear if every production is P is of the form
$A \rightarrow a$ or $A \rightarrow B\ a$
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 Grammer: A context-free grammer
G = (V, T, P, S)
is called regular, if it is either left-linear or it is right-linear.

Example 2.6.1.1: The regular language $L=\{a^n: n \geq 1\}$ over $T=\{a\}$ has the regular grammer given by

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

We have already studied the context-free grammars and context-free languages (Types2 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 Grammer A grammer
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 grammer is called a context-sensitive language

Example 2.6.1.2:

We just mentioned, without actually producing a grammer, 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 $x=a_1\ a_2 \ldots a_n$ in the following manner

..	..	..	$\langle$	a_1	a_2	…	a_n	$\rangle$

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 scam the cells containing $\rangle$ and $\langle$ but, these cells can not be written into

(iii) Tape head cannot move to or seam any cell right of 〉 and any cell to the left

of 〈

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 Grammer:
A grammer
G = (V, T, P, S),
is said to be phrase-structure unrestricted grammer, if P consists of productions of the form
$\alpha \rightarrow B$,
where

(i) α, B are strings over But 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 $\sum$, for which there is a TM M such that for each string $x \in \sum^*$, 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 Grammers/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 time.

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 grammers (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 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

$\delta(q_2, \#, \#) = (q_3, (\#, N), (\#, L))$
$\delta(q_3, \#, \overline{\#}) = (q_3, (\overline{\#}, R), (\#, L))$

(Copying symbols of Tape2 to Tape1 in reverse order)

$\delta(q_3, \#, \#) = (Halt, \#, \#)$

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, Σ, Γ, δ, q_0, h}, with

Q = {q_0, q_1, h}, Σ = {a, b}, Γ = { a, b, #} and δ being given by the following Transition Diagram

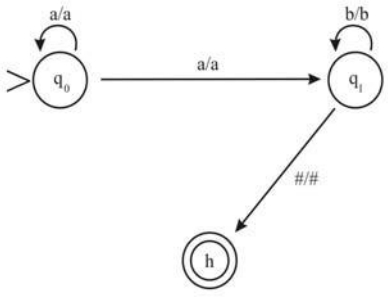


Fig 1.8.1

(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 b.

(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 $\sum^* \sim L$, **we construct a 3-tape TM M as follows:**

Let $M = \{Q, \sum, \Gamma, \delta, q_0, h\}$, $\sum = \{a, b, c\}$

$\Gamma = \{a, b, c\#\}$

The sequence of steps for defining δ for the required TM is, as given below

Step 1: Write any given string w over $\sum = \{a, b, c\}$ 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:

$\delta(q_0, (\#,\#,\#)) = (q1, (\#,\#,\#), (R, R, R))$

$\delta(q_1, (x, \#, \#)) = (q_1, (x,x,x), (R,R,R))$ for $x \in \sum$

$\delta(q_1, (\#, \#,\#)) = (q_2, (\#,\#,\#), (L,L,L))$

$\delta(q_2, (\#,\#,\#)) = (h (\#,\#,\#) (N,N,N)$

(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 H_1 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 onTape3. 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 o fany a on Tape 1.

Step 4: Next we match number of a'& on Tape1, to number of b's on Tape2 and number of c's on Tape3 through the following moves:

$\delta(q_4, (a,b,c)) = (q_4, (a,b,c)), (L,L,L)$ 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 $\sum^* \sim L$.

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)