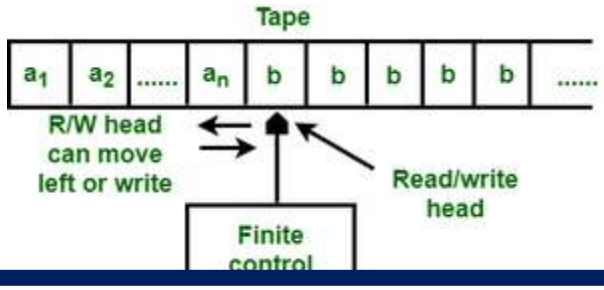
Unit3 TOC

Information Theory and Coding (Guru Gobind Singh Indraprastha University)

UNIT -3

Turing Machine

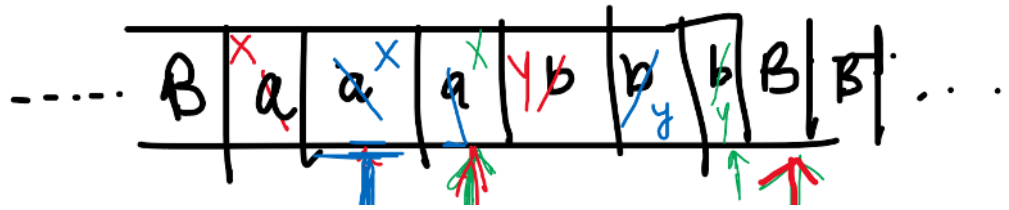
- **Turing Machine** was invented by Alan Turing in 1936 and it is used to accept Recursive Enumerable Languages (generated by Type-0 Grammar).
- A **Turing machine** consists of a tape of infinite length on which read and writes operation can be performed. The tape consists of infinite cells on which each cell either contains input symbol or a special symbol called blank. It also consists of a head pointer which points to cell currently being read and it can move in both directions.

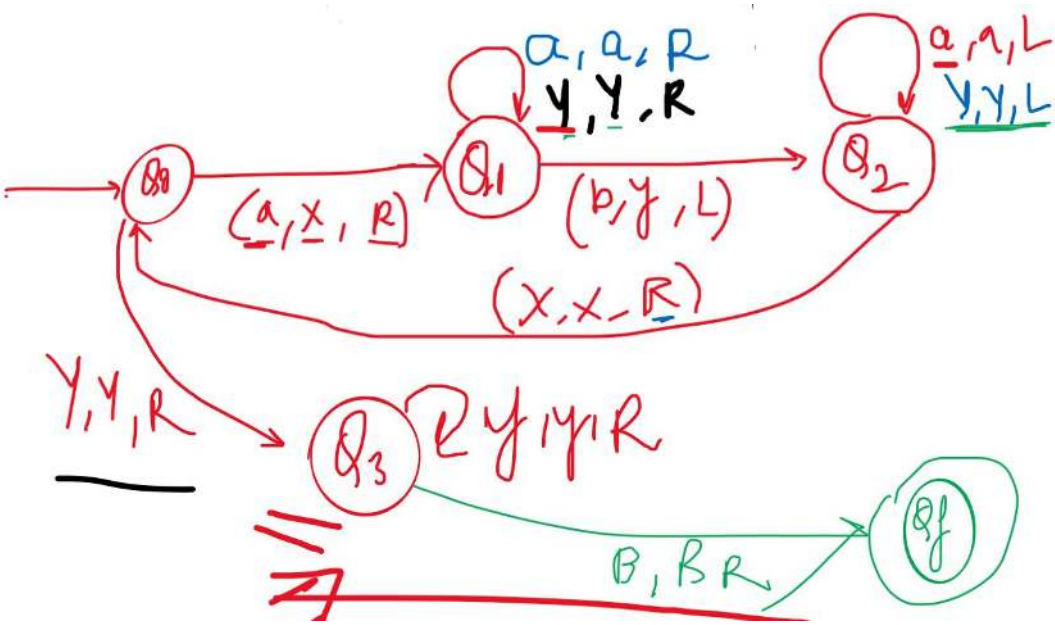


A TM is expressed as a **7-tuple (Q, T, B, ∑, δ, q0, F) where**:

- **Q** is a finite set of states
- **T** is the tape alphabet (symbols which can be written on Tape)
- **B** is blank symbol (every cell is filled with B except input alphabet initially)
- ∑ is the input alphabet (symbols which are part of input alphabet)
- **δ** is a transition function which maps Q × T → Q × T × {L,R}. Depending on its present state and present tape alphabet (pointed by head pointer), it will move to new state, change the tape symbol (may or may not) and move head pointer to either left or right.
- **q0** is the initial state
- **F** is the set of final states. If any state of F is reached, input string is accepted.

Question 1: Turing machine for $a^n b^n \mid n \geq 1$





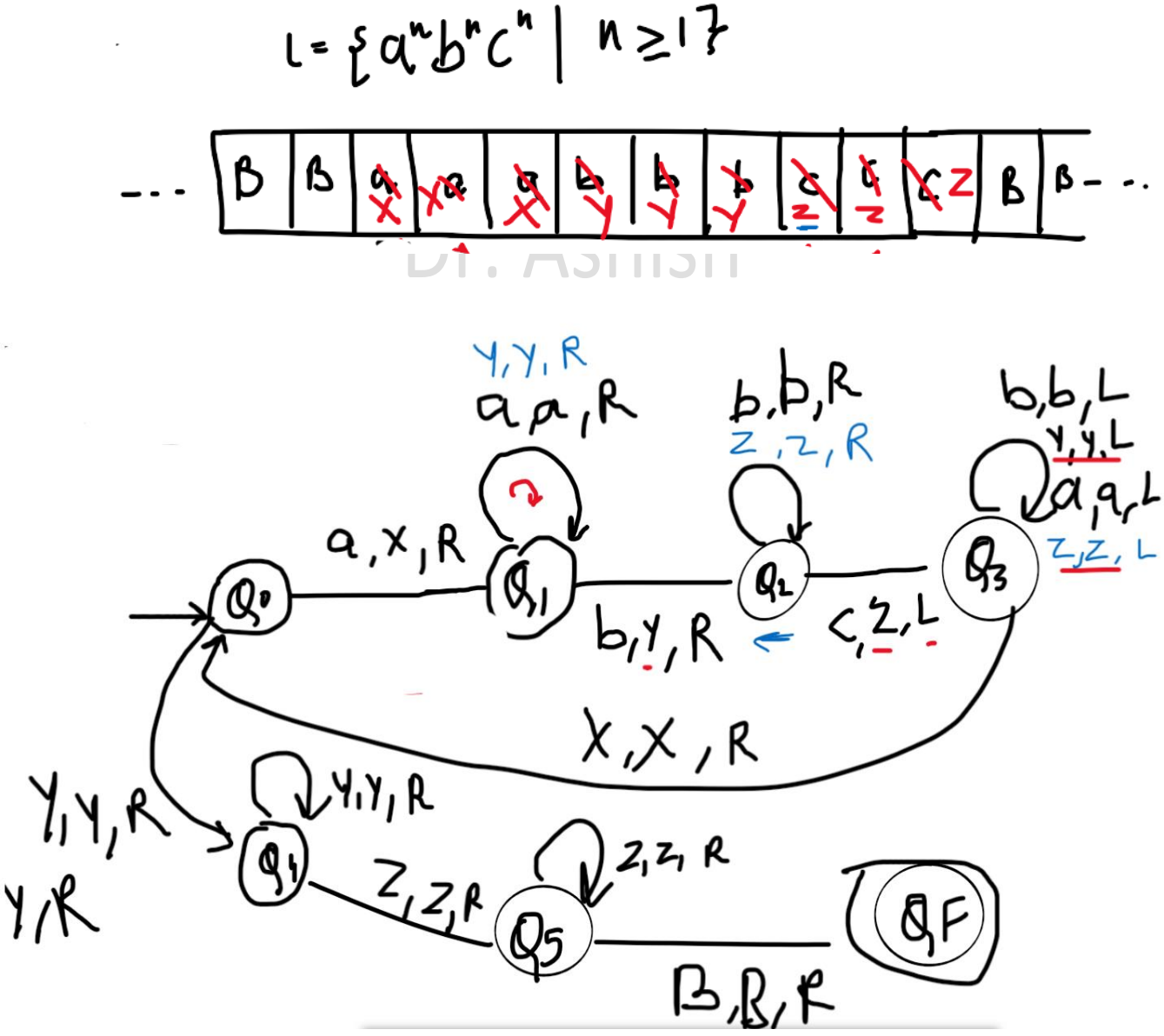
Approach for $a^n b^n \mid n \geq 1$

	a	b	x	Y	B
q_0	(q_1, X, R)			q_2, Y, R	
q_1	q_1, a, R	q_2, Y, L		q_1, Y, R	
q_2	q_2, a, L		$q_0 X, R$	q_2, Y, L	
q_3				q_3, Y, R	Accept. Halt.

Question 2: Turing machine for $a^n b^n c^n \mid n \geq 1$

Approach for $a^n b^n c^n \mid n \geq 1$

1. Mark 'a' then move right.
2. Mark 'b' then move right
3. Mark 'c' then move left
4. Come to far left till we get 'X'
5. Repeat above steps till all 'a', 'b' and 'c' are marked
6. At last if everything is marked that means string is accepted.



Turing Church's Thesis

Turing Machines: a lesson from history

SUR LES

PROBLÈMES FUTURS DES MATHÉMATIQUES,

PAR M. DAVID HILBERT (Göttingen),

TRADUITE PAR M. L. LAUGEL (1).

Qui ne soulèverait volontiers le voile qui nous cache l'avenir afin de jeter un coup d'œil sur les progrès de notre Science et les secrets de son développement ultérieur durant les siècles futurs? Dans ce champ si fécond et si vaste de la Science mathématique, quels seront les buts particuliers que tenteront d'atteindre les guides de la

Hilbert's lecture from the Paris Proceedings

Hilbert's problems

- In 1900, David Hilbert listed out 23 problems as challenges for 20th century at the Int. Cong. of Mathematicians in Paris.
- 10th problem: Devise an algorithm (a process doable using a finite no. of operations) to test if a given polynomial has integral roots.

The Church-Turing Thesis

Alonso Church
(1903–1995)

Alan Turing
(1912–1954)

the Church-Turing Thesis:

"a problem can be solved by an algorithm iff it can be solved by a Turing Machine"

Turing Machines → implement → algorithms

all algorithmically solvable problems can be solved by a Turing Machine

The Church-Turing Thesis

- The Church-Turing Thesis provides the definition of algorithm necessary to resolve Hilbert's tenth problem.

The Church-Turing Thesis

Our intuitive understanding of algorithms coincides with Turing machine algorithms.

- After studying the equivalence of several such models, they made the **Church-Turing thesis.**

- "a function is realistically computable if and only if it is computable by a Turing machine".

- He also realized that **Turing machines and λ-calculus are equivalent** models of computation

Formalisms for Algorithms

By the 1930s the emphasis was on formalising algorithms

Alan Turing, at Cambridge, devised an abstract machine now called a Turing Machine to define/represent algorithms

Alonso Church, at Princeton, devised the Lambda Calculus which formalises algorithms as functions..more later in the course.

neither knew of the other's work in progress..both published in 1936

the demonstrated equivalence of their formalisms strengthened both their claims to validity, expressed as the Church-Turing Thesis.

The Church-Turing Thesis

– The Church-Turing Thesis provides the definition of algorithm necessary to resolve Hilbert's tenth problem.

The Church-Turing Thesis

Our intuitive understanding of algorithms coincides with Turing machine algorithms.

– Thus, Turing machines capture our intuitive idea of computation.
– Indeed, we are more interested in algorithms themselves. Once we believe that TMs precisely capture algorithms, we will shift our focus to algorithms rather than low-level descriptions of TMs.

Church-Turing thesis

- The definition came in the 1936 papers of A. Church and A.Turing.
- Church used a notational system called λ -calculus to define algorithms.
- Turing did it with his 'machines'.
- These two definitions were shown to be equivalent.
- This connection between the informal notion of algorithm and the precise definition has

come to be called the Church-Turing thesis.

Intuitive notion of algorithms == Turing machine of algorithms

- This thesis provides the definition of algorithm necessary to resolve Hilbert's tenth problem.
- In 1970, Yuri Matijasevich showed that no algorithm exists for testing whether a polynomial has integral roots.
- Later we will see the techniques that form the basis for proving that this and other problems are algorithmically unsolvable

More on the Church-Turing Thesis

Lambda Calculus

- In 1936, Church introduced Lambda Calculus as a formal description of all computable functions.
- Independently, Turing had introduced his A-machines in 1936 too.
- Turing also showed that his A-machines were equivalent to Lambda Calculus of Church.

- So, can a Turing machine do everything? In other words are there algorithms to solve every question. Godel's incompleteness result asserts otherwise.

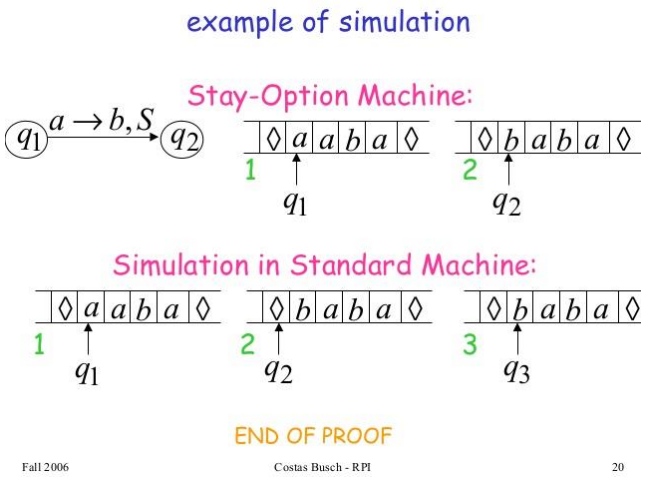
Equivalence Of Turing Machine

To say that a computational device is Turing equivalent means that it can do anything a Turing machine can. Many machines that might be thought to have more computational capability than a simple universal Turing machine can be shown to have no more power. They might compute faster, perhaps, or use less memory, or their instruction set might be smaller, but they cannot compute more powerfully (i.e. more mathematical functions).

Variants of Turing Machine

Turing Machine with stay option:

- In standard Turing machine, the read write head must move either to the right or to the left. In Turing machine with stay option we have the option to have the read read-write head stay in place after rewriting the cell content. Thus, we can define a Turing machine with stay-option by replacing δ by
- δ : Q x Γ → Q x Γ x { L,R,S}
- with the interpretation that S signifies no movement of the read write hand. This option does not extend the power of automation.



Multiple Track Turing Machine:

- A k-tack Turing machine (for some k>0) has k-tracks and one R/W head that reads and writes all of them one by one.
- A k-track Turing Machine can be simulated by a single track Turing machine

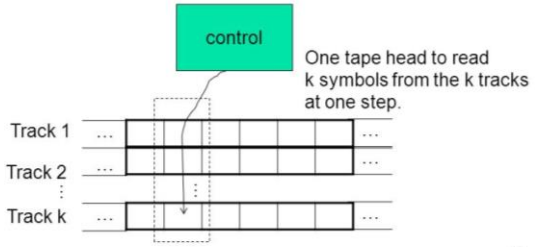
Multi-track TM ==> TM (proof)

- *For every M', there is an M s.t. L(M')=L(M).*
  - M = (Q, ∑, Γ, δ, q_0,[B,B,…],F)
  - Where:
    - Q = Q'
    - ∑ = ∑ ' x ∑ ' x … (k times for k-track)
    - Γ = Γ' x Γ' x … (k times for k-track)
    - $q_0 = q'_0$
    - F = F'
    - $\delta(q_i,[a_1,a_2,\ldots a_k]) = \delta'(q_i, <a_1,a_2,\ldots a_k>)$
- Multi-track TMs are just a different way to represent single-track TMs, and is a matter of design convenience.

Main idea: Create one composite symbol to represent every combination of k symbols

Multi-track Turing Machines

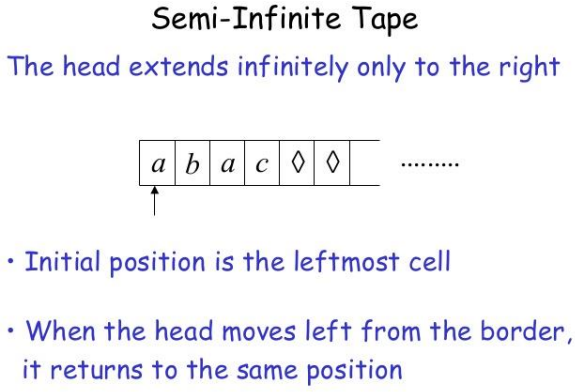
- TM with multiple tracks, but just one unified tape head



18

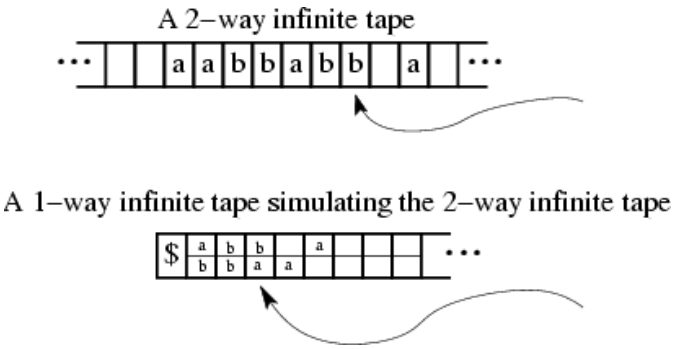
Turing Machine with Semi-infinite tape

- We know that Turing machine has an infinite input tape with extends in both the directions (left and right) infinitely. So now if we restrict it to extend only in one direction and not in both the directions, i.e., we make the tape to be semi-infinite, then also the number of languages accepted by the Turing machine remains same.



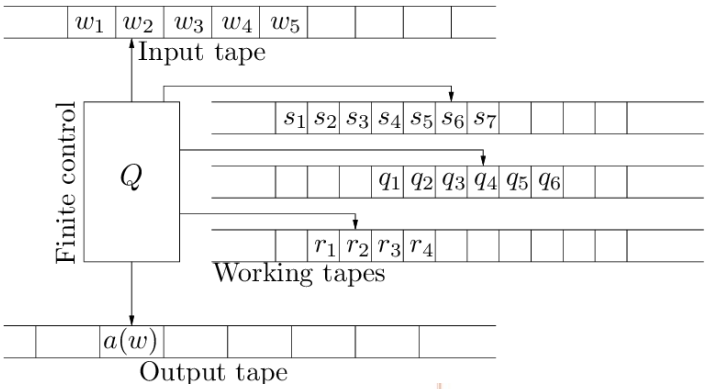
Two-way infinite Tape Turing Machine:

- Infinite tape of two-way infinite tape Turing machine is unbounded in both directions left and right.
- Two-way infinite tape Turing machine can be simulated by one-way infinite Turing machine (standard Turing machine).



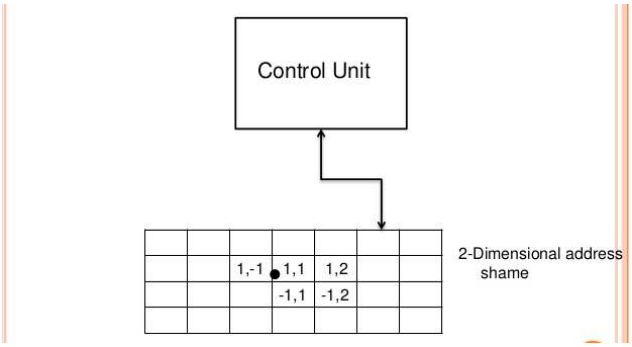
Multi-tape Turing Machine:

- It has multiple tapes and controlled by a single head.
- The Multi-tape Turing machine is different from k-track Turing machine but expressive power is same.
- Multi-tape Turing machine can be simulated by single-tape Turing machine.
  $\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{ L, R \}^k$



Multi-dimensional Tape Turing Machine:

- It has multi-dimensional tape where head can move any direction that is left, right, up or down.

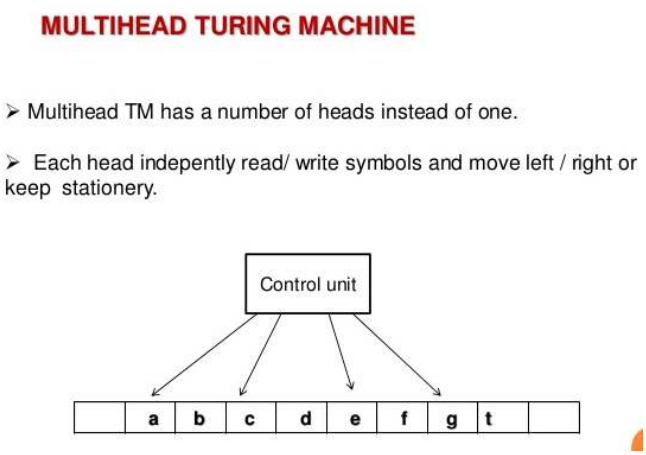


- Multi-dimensional tape Turing machine can be simulated by one-dimensional Turing machine

  $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R,U,D\}$

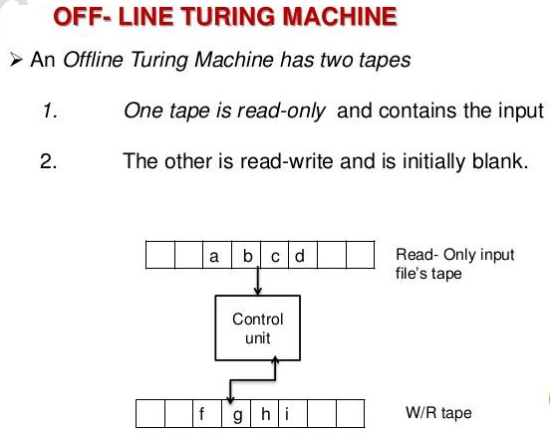
Multi-head Turing Machine:

- A multi-head Turing machine contains two or more heads to read the symbols on the same tape.
- In one step all the heads sense the scanned symbols and move or write independently.
- Multi-head Turing machine can be simulated by single head Turing machine.
-



Offline Turing machine

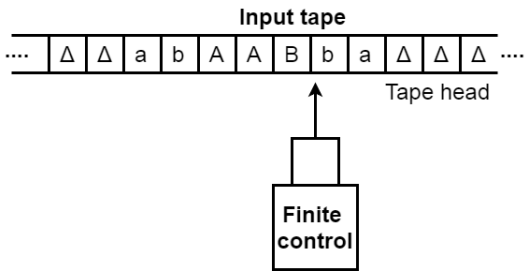
- In standard Turing machine both the input and output are present on the tape, the head has the authority to move across the input and can change or modify the input, if we don't want to modify the input we can provide the input in separate file, which is read only then the head cannot make changes to it.
- If the Turing machine wants to modify the input, the input need to be copied on the tape and the changes can be made by the head but still the input file remains unchanged as changes are made in the tape and not in the file. By doing such modification to Turing machine still the number of languages accepted by the Turing machine remains the same.



Jumping Turing Machine

- The standard Turing machine's head can move only one step to the right or left, but in case of jumping Turing machine the head can move not only just one step to the left or right but it can move more the one, i.e., 2, 3, 4, 5, 6, … so on, or it can jump to any cell to the right or left of the input tape.
- $\delta: Q \times \Gamma \dashrightarrow Q \times \Gamma \times \{L,R\} \times \{n\}$

n, is the number of steps that we wish to move to the right or left. But still the languages accepted by Turing machine remains the same.



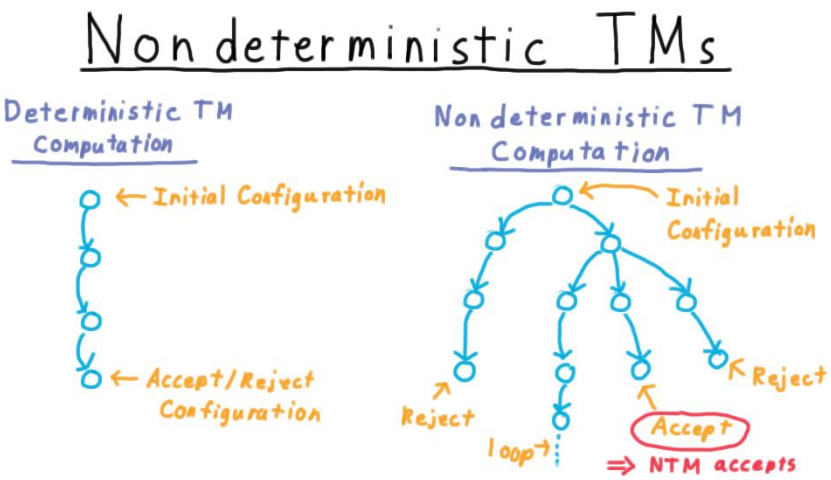
- ptr = ptr+n;
- ptr = ptr-n

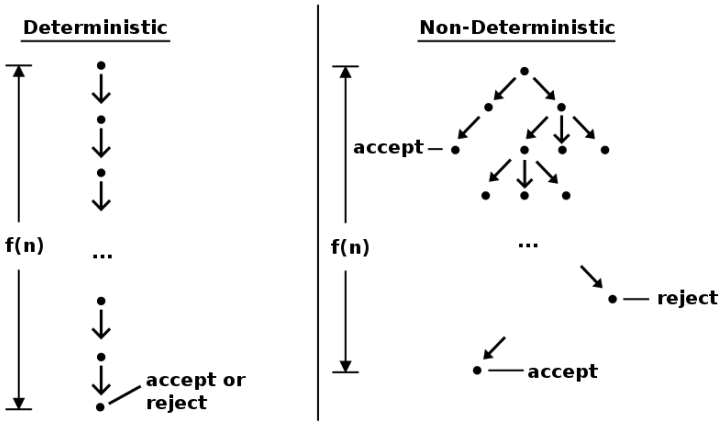
Non-deterministic Turing Machine:

- A non-deterministic Turing machine has a single, one way infinite tape.
- For a given state and input symbol has at least one choice to move (finite number of choices for the next move), each choice several choices of path that it might follow for a given input string.

- A non-deterministic Turing machine is equivalent to deterministic Turing machine.

$\delta : Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L,R\}}$



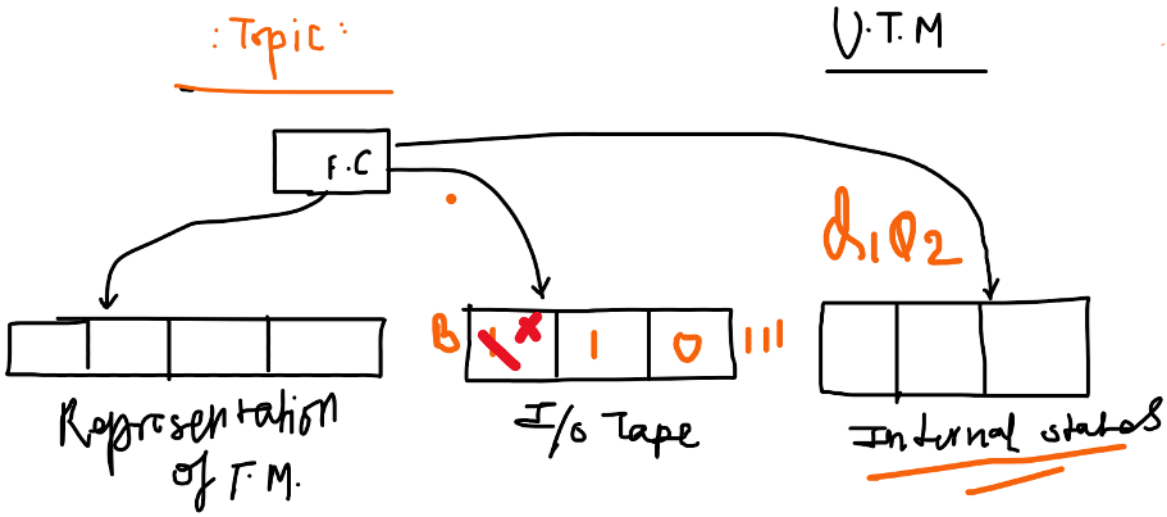


Universal Turing machine

- A Turing machine is said to be universal Turing machine if it can accept:
    - The input data, and
    - An algorithm (description) for computing.
- This is precisely what a general-purpose digital computer does. A digital computer accepts a program written in high level language. Thus, a general purpose Turing machine will be called a universal Turing machine if it is powerful enough to simulate the behavior of any digital computer, including any Turing machine itself.
- More precisely, a universal Turing machine can simulate the behavior of an arbitrary Turing machine over any set of input symbols. Thus, it is possible to create a single machine that can be used to **compute any computable sequence.**
- If this machine is supposed to be supplied with the tape on the beginning of which is written the input string of quintuple separated with some special symbol of some computing machine M, then the universal Turing machine U will compute the same strings as those by M.
- The model of a Universal Turing machine is considered to be a theoretical breakthrough that led to the concept of stored programmer computing device.
- Designing a general purpose Turing machine is a more complex task. Once the transition of Turing machine is defined, the machine is restricted to carrying out one particular type of computation.
- By modifying our basic model of a Turing machine, we can design a universal Turing machine. The modified Turing machine must have a large number of states for stimulating even a simple behaviour. **We modify our basic model by:**
    - Increase the number of read/write heads
    - Increase the number of dimensions of input tape
    - Adding a special purpose memory
- All the above modification in the basic model of a Turing machine will almost speed up the operations of the machine can do.
- A number of ways can be used to explain to show that Turing machines are useful models of real computers. Anything that can be computed by a real computer can also be computed by a Turing machine. A Turing machine, for example can simulate any type of functions used in programming language. Recursion and parameter passing are some typical examples. A Turing machine can also be used to simplify the statements of an algorithm.

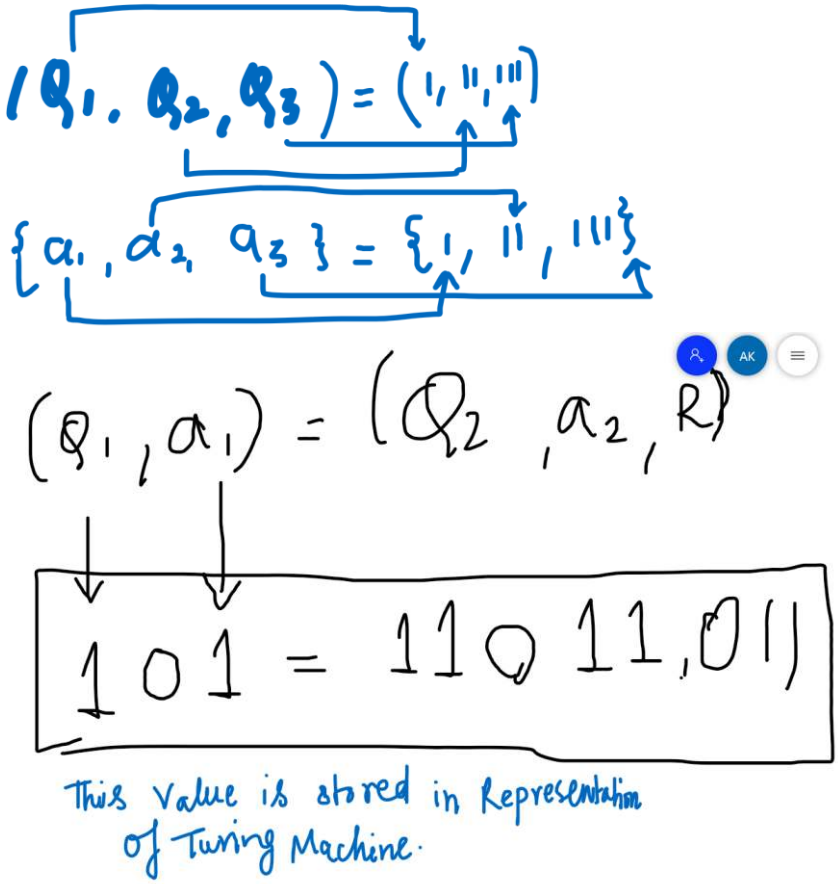
- A Turing machine is not very capable of handling it in a given finite amount of time. Also, Turing machines are not designed to receive unbounded input as many real programmers like word processors, operating system, and other system software.

Example of Universal Turing Machine



Mapping of the states and Input tape symbol with Binary digits

{L-1 , R-11 }



Recursive and Recursive Enumerable Languages in TOC

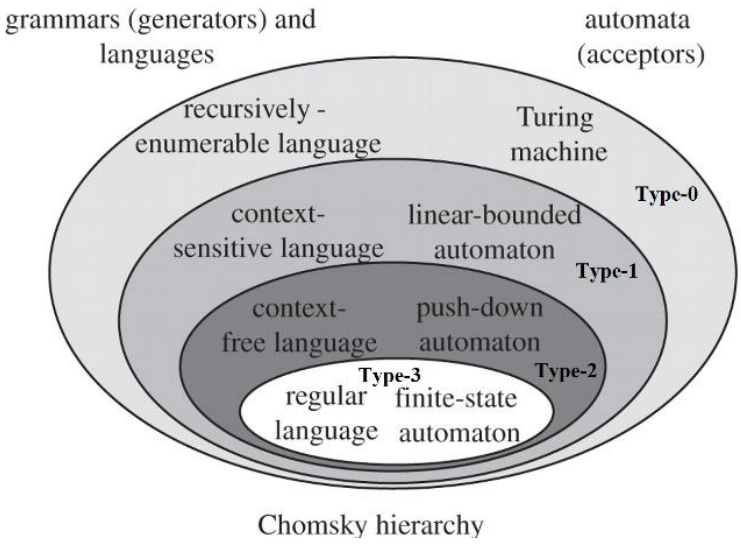
Recursive Enumerable (RE) or Type -0 Language

- RE languages or type-0 languages are generated by type-0 grammars.
- An RE language can be accepted or recognized by Turing machine which means it will enter into final state for the strings of language and may or may not enter into rejecting state for the strings which are not part of the language. It means TM can loop forever for the strings which are not a part of the language. **RE languages are also called as Turing recognizable languages.**

Recursive Language (REC)

A recursive language (subset of RE) can be decided by Turing machine which means it will enter into final state for the strings of language and rejecting state for the strings which are not part of the language.

e.g.; L= $\{a^n b^n c^n | n>=1\}$ is recursive because we can construct a turing machine which will move to final state if the string is of the form $a^n b^n c^n$ else move to non-final state. So the TM will always halt in this case. REC languages are also called as **Turing decidable languages**. The relationship between RE and REC languages can be shown in Figure 1.



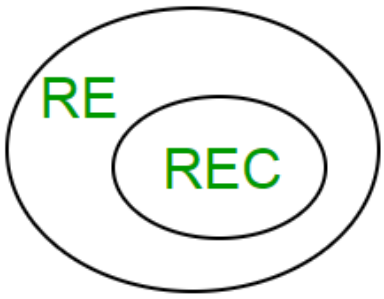


Figure 1

Closure Properties of Recursive Languages

- **Union**: If L1 and If L2 are two recursive languages, their union **L1∪L2 will also be recursive** because if TM halts for L1 and halts for L2, **it will also halt for L1∪L2.**

- **Concatenation:** If L1 and If L2 are two recursive languages**, their concatenation L1.L2 will also be recursive.** For Example:

  $L1= \{a^n b^n c^n | n>=0\}$

  $L2= \{d^m e^m f^m | m>=0\}$

  L3= L1.L2

  $= \{a^n b^n c^n d^m\ e^m f^m | m>=0$ and $n>=0\}$ is also recursive.

L1 says n no. of a's followed by n no. of b's followed by n no. of c's. L2 says m no. of d's followed by m no. of e's followed by m no. of f's. Their concatenation first matches no. of a's, b's and c's and then matches no. of d's, e's and f's. So it can be decided by TM.

- **Kleene Closure:** If L1is recursive, its kleene closure **L1* will also be recursive**. For Example:

  $L1= \{a^n b^n c^n | n>=0\}$

  $L1^*= \{ a^n b^n c^n || n>=0\}^*$ is also recursive.

- **Intersection and complement**: If L1 and If L2 are two recursive languages, **their intersection L1 ∩ L2 will also be recursive.** For Example:

  $L1= \{a^n b^n c^n dm | n>=0$ and $m>=0\}$

  $L2= \{a^n b^n c^n d^n | n>=0$ and $m>=0\}$

  L3=L1 ∩ L2

  $= \{ a^n b^n c^n d^n | n>=0\}$ will be recursive.

Similarly, complement of recursive language L1 which is ∑*-L1, will also be recursive.

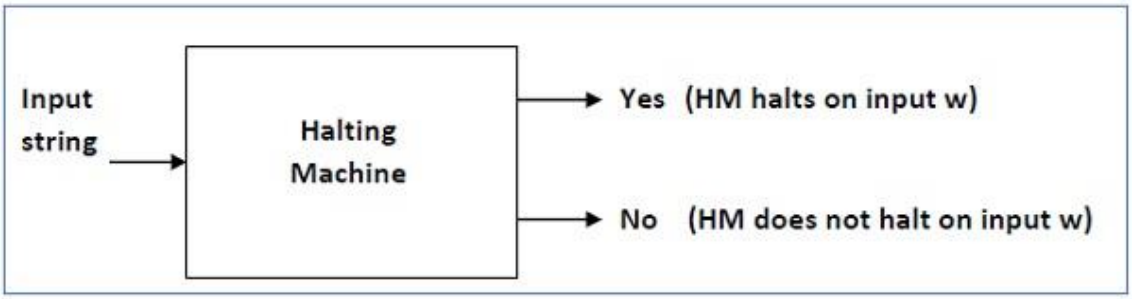
Note: As opposed to REC languages, RE languages are not closed under complement on which means complement of RE language need not be RE.

Turing Machine Halting Problem

Input − A Turing machine and an input string **w**.

Problem − Does the Turing machine finish computing of the string **w** in a finite number of steps? The answer must be either **yes or no.**

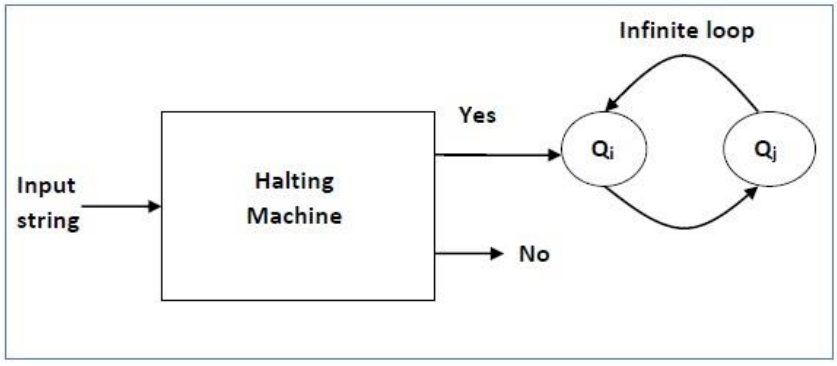
Proof − At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a **Halting machine** that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as 'yes', otherwise as 'no'. The following is the block diagram of a Halting machine −



Now we will design an **inverted halting machine (HM)'** as −

- If **H** returns YES, then loop forever.
- If **H** returns NO, then halt.

The following is the block diagram of an 'Inverted halting machine' −



Further, a machine **$(HM)_2$** which input itself is constructed as follows −

- If $(HM)_2$ halts on input, loop forever.
- Else, halt.

Here, we have got a contradiction. Hence, **the halting problem is undecidable.**

Decidable and Undecidable problems in Theory of Computation

A problem is said to be **Decidable** if we can always construct a corresponding **algorithm** that can answer the problem correctly. We can intuitively understand Decidable problems by considering a simple example.

Suppose we are asked to compute all the prime numbers in the range of 1000 to 2000. To find the **solution** of this problem, we can easily devise an algorithm that can enumerate all the prime numbers in this range.

Now talking about Decidability in terms of a Turing machine, a problem is said to be a Decidable problem if there exists a corresponding Turing machine which **halts** on every input with an answer- **yes or no**. It is also important to know that these problems are termed as **Turing Decidable** since a Turing machine always halts on every input, accepting or rejecting it.

- **Semi- Decidable Problems –**Semi-Decidable problems are those for which a Turing machine halts on the input accepted by it but it can either halt or loop forever on the input which is rejected by the Turing Machine. Such problems are termed as **Turing Recognisable** problems.

Examples – We will now consider few important **Decidable problems**:

- Are two **regular** languages L and M **equivalent**?
- We can easily check this by using Set Difference operation. L-M =Null and M-L =Null. Hence (L-M) U (M-L) = Null, then L,M are equivalent.
- Membership of a CFL?

  We can always find whether a string exists in a given CFL by using an algorithm based on dynamic programming.
- Emptiness of a CFL

  By checking the production rules of the CFL we can easily state whether the language generates any strings or not.

- **Undecidable Problems –**

The problems for which we can't construct an algorithm that can answer the problem correctly in finite time are termed as Undecidable Problems. These problems may be partially decidable but they will never be decidable. That is there will always be a condition that will lead the **Turing Machine into an infinite loop without providing an answer at all.**

We can understand Undecidable Problems intuitively by considering **Fermat's Theorem**, a popular Undecidable Problem which states that no three positive integers a, b and c for any n>2 can ever satisfy the equation: a^n + b^n = c^n.

If we feed this problem to a Turing machine to find such a solution which gives a contradiction then a Turing Machine might run forever, to find the suitable values of n, a, b and c. But we are always unsure whether a contradiction exists or not and hence we term this problem as an **Undecidable Problem**.

Examples – These are few important **Undecidable Problems**:

- Whether a CFG generates all the strings or not?

  As a CFG generates infinite strings, we can't ever reach up to the last string and hence it is Undecidable.
- **Whether two CFG L and M equal?**

  Since we cannot determine all the strings of any CFG, we can predict that two CFG are equal or not.
- **Ambiguity of CFG?**

  There exist no algorithm which can check whether for the ambiguity of a CFL. We can only check if any particular string of the CFL generates two different parse trees then the CFL is ambiguous.
- Is it possible to convert a given ambiguous CFG into corresponding non-ambiguous CFL?

  It is also an Undecidable Problem as there doesn't exist any algorithm for the conversion of an ambiguous CFL to non-ambiguous CFL.
- Is a language Learning which is a CFL, regular?

  This is an Undecidable Problem as we can not find from the production rules of the CFL whether it is regular or not.

Some more **Undecidable Problems** related to Turing machine:

- **Membership** problem of a Turing Machine?
- **Finiteness** of a Turing Machine?
- **Emptiness** of a Turing Machine?
- Whether the language accepted by Turing Machine is regular or CFL?

AKASH TOMAR
35313302719
CSE-4B

Theory of computation Assignment

Topic:- Examples of undecidable problems

Undecidable Problems –

The problems for which we can't construct an algorithm that can answer the problem correctly in finite time are termed as Undecidable Problems. These problems may be partially decidable but they will never be decidable. That is there will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer at all.

Examples:-

1) Whether two CFG L and M equal?

- Since we cannot determine all the strings of any CFG, we can predict that two CFG are equal or not.

2) Ambiguity of CFG?

- There exist no algorithm which can check whether for the ambiguity of a CFL. We can only check if any particular string of the CFL generates two different parse trees then the CFL is ambiguous.

3) Is it possible to convert a given ambiguous CFG into corresponding non-ambiguous CFL?

AKASH TOMAR
35313302719
CSE-4B

➢ It is also an Undecidable Problem as there doesn't exist any algorithm for the conversion of an ambiguous CFL to non-ambiguous CFL.

4) Is a language Learning which is a CFL, regular?

➢ This is an Undecidable Problem as we can not find from the production rules of the CFL whether it is regular or not.

5) Whether a CFG generates all the strings or not?

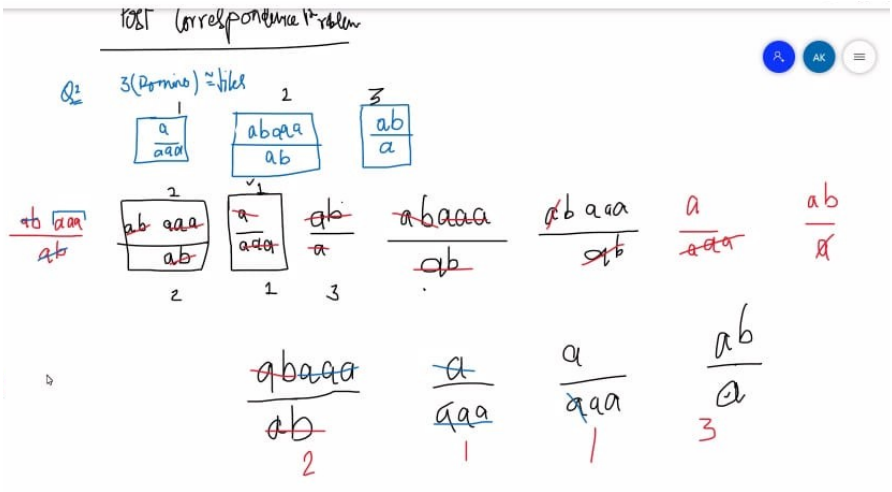
➢ As a CFG generates infinite strings, we can't ever reach up to the last string and hence it is Undecidable.

UNDECIDABILITY IN POST CORRESPONDENCE PROBLEM

Post Correspondence Problem is a popular **undecidable problem** that was introduced by Emil Leon Post in 1946. It is simpler than Halting Problem.

In this problem we have N number of Dominos (tiles). The aim is to arrange tiles in such order that string made by Numerators is same as string made by Denominators.
In simple words, lets assume we have two lists both containing N words, aim is to find out concatenation of these words in some sequence such that both lists yield same result.

AKASH TOMAR
35313302719
CSE-4B



We can try unlimited combinations but none of combination will lead us to solution, thus this problem does not have solution.

So , the problem is undecidable.

The Halting Problem

Given a Program, WILL IT HALT?

Given a Turing machine, will it halt when run on some particular given input string.

Given some program written in some language (Java / C / C++ / python etc) will it ever get into an infinite loop or will it always terminate?

Answer :-

:- • In General we can't always know.

• The Best we can do is run the program and see whether it halts.

• For many programs we can see that it will always halt or sometimes loop.

But for programs in general the question is undecidable.

If it is undecidable so there is no Turing machine for this problem.

From the church Turing thesis we have already studied that anything that is computable or anything that has an algorithm can be designed using a Turing machine.

But if we think it is in reverse way, we can say that if there is a Turing machine for a particular task then that will have an algorithm or that is computable.

Here we said that the problem is undecidable and if the problem is undecidable so there is no turing machine for that and from church thesis if there is no Turing machine so there is no algorithm for that.

§

Undecidability of the Halting Problem

Given a program, will it Halt?

Can we design a machine which if given a program can find out or decide if that program will always halt or not halt on a particular input?

Let us assume that it can halt:

Machine named ⇒ H
which takes two arguments,
first Program P, and Second Input I.

PAGE NO.
DATE / /

This machine H on taking Program P and input I as argument Tells us either the program P will halt or not on taking input I.

H (P, I)
- Halt
- Not Halt

This allow us to write another Program:

C(X)
if { H(X, X) == Halt }
    Loop forever;
else
    Return;
    ↳ this statement actually halts the program on getting not halt in H(X, X).

C function has two part ifpart and else part.

If we run C on itself.

C(C)

H (C, C) == Halt
↓
Not Halt.

H (C, C) == Not Halt
↓
return / Halt

It is a contradiction. It is undecidable.