KDK College of Engineering
Department of Information Technology
Subject - Theory of Computation (IT403T)
Unit – 3,4,5 Theory

UNIT III:

Closure properties of context free languages

Context-free languages are **closed** under –

- Union
- Concatenation
- Kleene Star operation

Union

Let L_1 and L_2 be two context free languages. Then $L_1 \cup L_2$ is also context free.

Example

Let $L_1 = \{ a^n b^n , n > 0\}$. Corresponding grammar G_1 will have P: S1 → aAb|ab

Let $L_2 = \{ c^m d^m , m \geq 0\}$. Corresponding grammar G_2 will have P: S2 → cBb| ε

Union of L_1 and L_2, $L = L_1 \cup L_2 = \{ a^n b^n \} \cup \{ c^m d^m \}$

The corresponding grammar G will have the additional production S → S1 | S2

Concatenation

If L_1 and L_2 are context free languages, then L_1L_2 is also context free.

Example

Union of the languages L_1 and L_2, $L = L_1L_2 = \{ a^n b^n c^m d^m \}$

The corresponding grammar G will have the additional production S → S1 S2

Kleene Star

If L is a context free language, then L* is also context free.

Example

Let $L = \{ a^n b^n , n \geq 0\}$. Corresponding grammar G will have P: S → aAb| ε

Kleene Star $L_1 = \{ a^n b^n \}^*$

The corresponding grammar G_1 will have additional productions $S1 \rightarrow SS_1 \mid \varepsilon$

Context-free languages are **not closed** under –

Intersection – If L1 and L2 are context free languages, then L1 ∩ L2 is not necessarily context free.

Intersection with Regular Language – If L1 is a regular language and L2 is a context free language, then L1 ∩ L2 is a context free language.

Complement – If L1 is a context free language, then L1' may not be context free.

Union

Suppose we have grammars for two languages, with start symbols S and T. Rename variables as needed to ensure that the two grammars don't share any variables. Then construct a grammar for the union of the languages, with start symbol Z, by taking all the rules from both grammars and adding a new rule Z -> S | T.

Concatenation

Suppose we have grammars for two languages, with start symbols S and T. Rename variables as needed to ensure that the two grammars don't share any variables. Then construct a grammar for the union of the languages, with start symbol Z, by taking all the rules from both grammars and adding a new rule Z -> ST.

Star

Suppose that we have a grammar for the language L, with start symbol S. The grammar for L*, with start symbol T, contains all the rules from the original grammar plus the rule T -> TS | ε.

String reversal

Reverse the character string on the righthand side of every rule in the grammar.

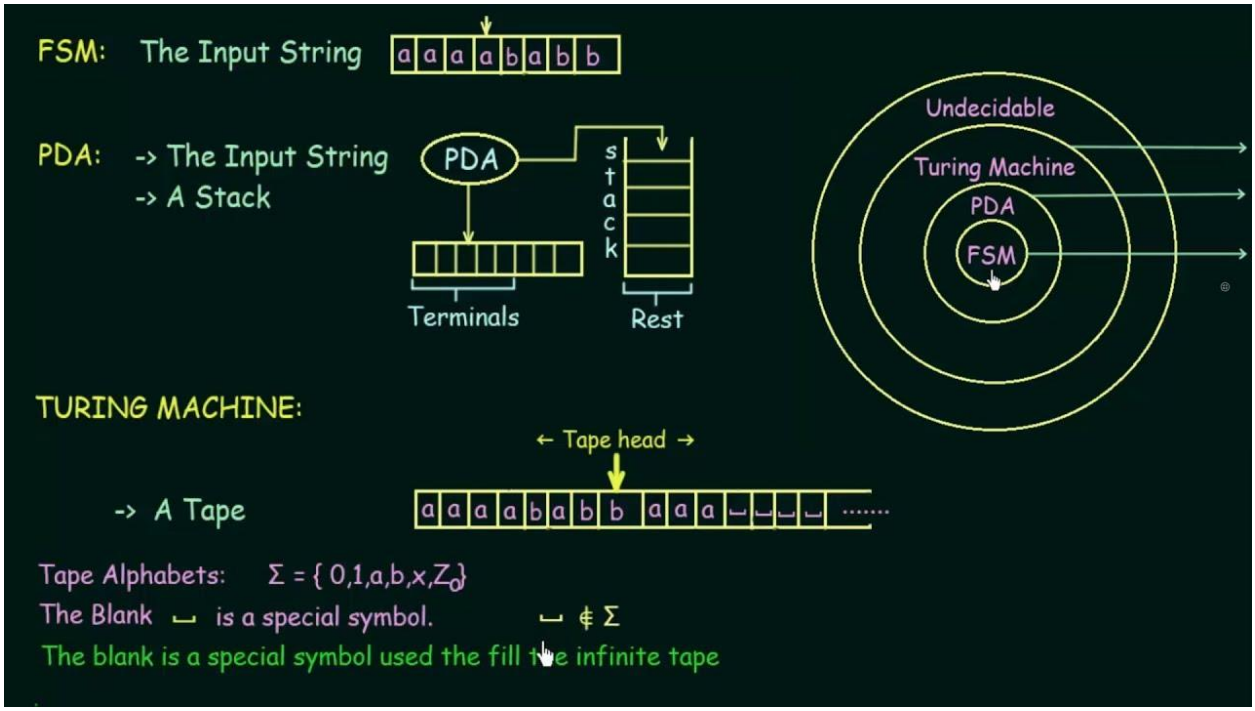
Homomorphism

Suppose that we have a grammar G for language L and a homomorphism h. To construct a grammar for h(L), modify the righthand side of every rule in G to replace each terminal symbol t with its image h(t) under the homomorphism.

Intersection with a regular language

The intersection of a context-free language and a regular language is always context-free. To show this, assume we have a PDA M accepting the context-free language and a DFA N accepting the regular language. Use the product construction to create a PDA which simulates both machines in parallel. This works because only M needs to manipulate the stack; N never touches the stack.

UNIT IV:



1. Turing Machine: Definition,

A Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars. It was invented in 1936 by Alan Turing.

Definition

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A TM can be formally described as a 7-tuple (Q, X, Σ, δ, q_0, B, F) where –

- **Q** is a finite set of states
- **X** is the tape alphabet
- **Σ** is the input alphabet
- **δ** is a transition function; δ : Q × X → Q × X × {Left_shift, Right_shift}.
- **q_0** is the initial state
- **B** is the blank symbol
- **F** is the set of final states

Comparison with the previous automaton

The following table shows a comparison of how a Turing machine differs from Finite Automaton and Pushdown Automaton.

Machine	Stack Data Structure	Deterministic?
Finite Automaton	N.A	Yes
Pushdown Automaton	Last In First Out(LIFO)	No
Turing Machine	Infinite tape	Yes

Example of Turing machine
Turing machine M = (Q, X, Σ, δ, q_0, B, F) with

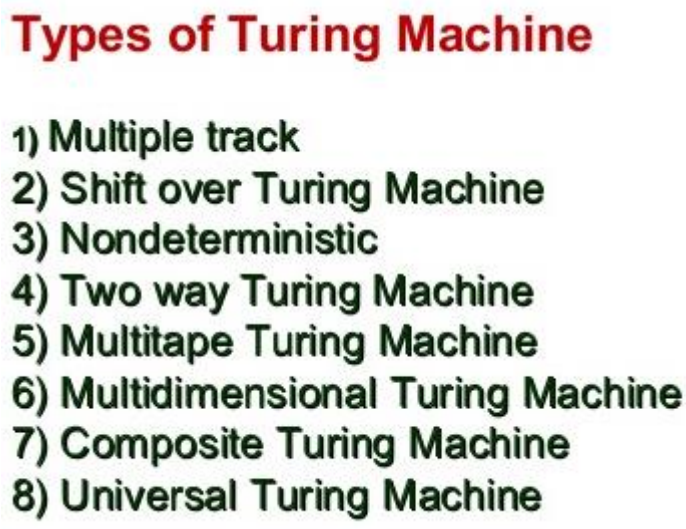
- Q = {q_0, q_1, q_2, q_f}
- X = {a, b}
- Σ = {1}
- q_0 = {q_0}
- B = blank symbol
- F = {q_f }

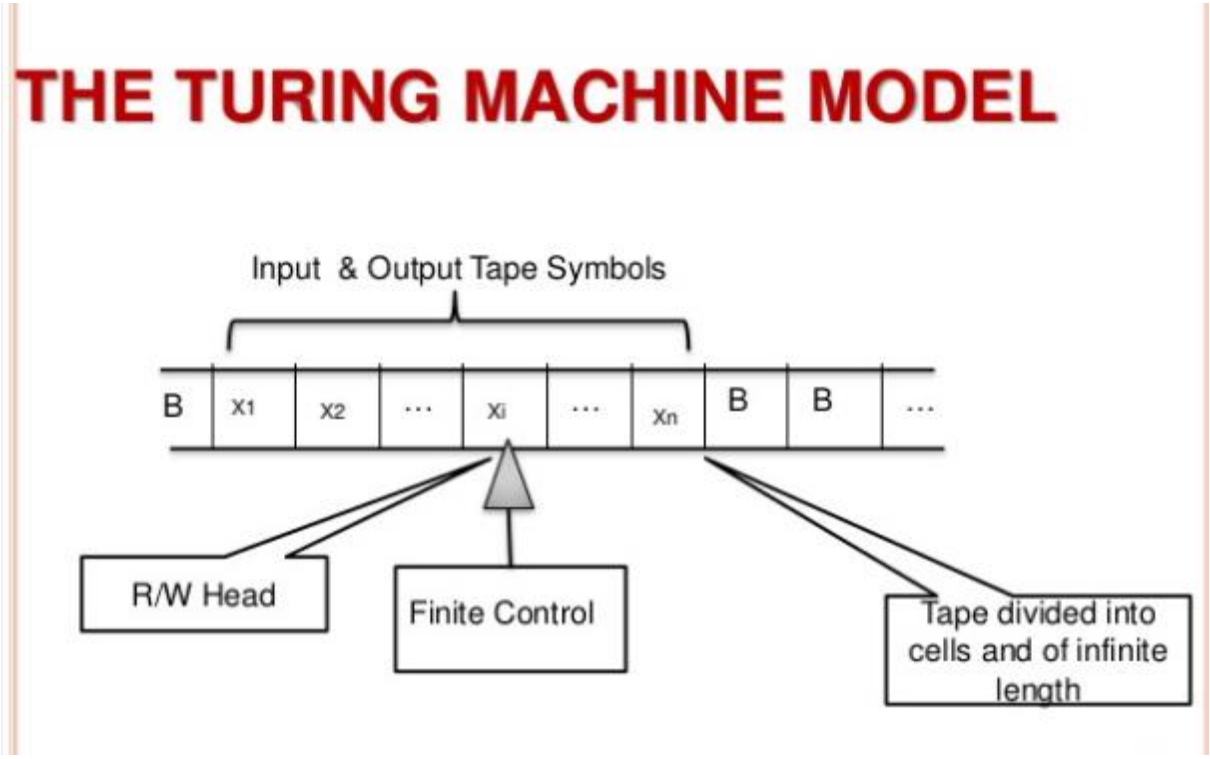
δ is given by –

Tape alphabet symbol	Present State 'q₀'	Present State 'q₁'	Present State 'q₂'
a	$1Rq_1$	$1Lq_0$	$1Lq_f$
b	$1Lq_2$	$1Rq_1$	$1Rq_f$

Here the transition $1Rq_1$ implies that the write symbol is 1, the tape moves right, and the next state is q_1. Similarly, the transition $1Lq_2$ implies that the write symbol is 1, the tape moves left, and the next state is q_2.

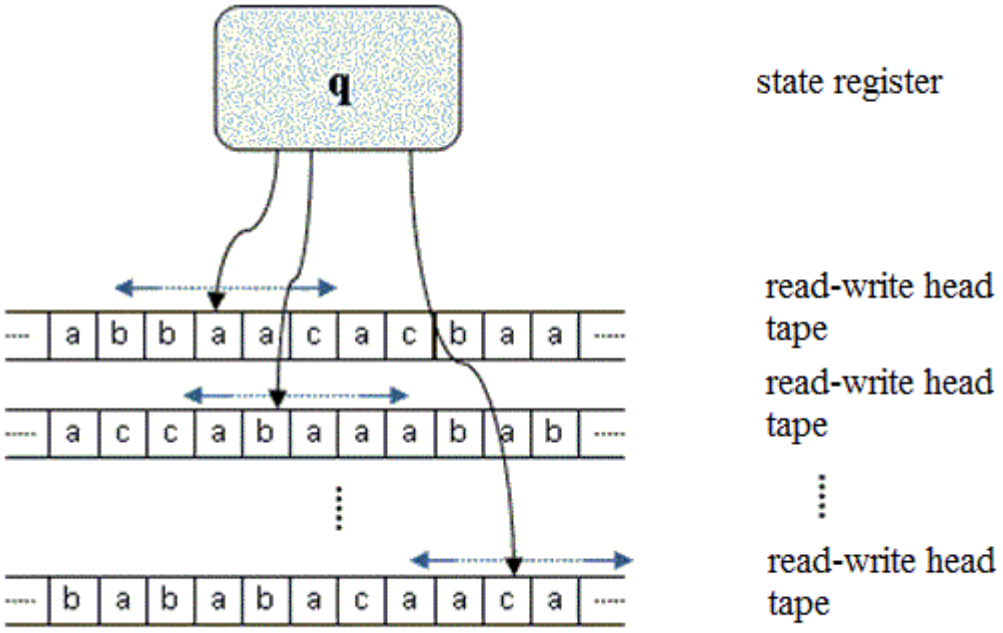
2. Model of TM

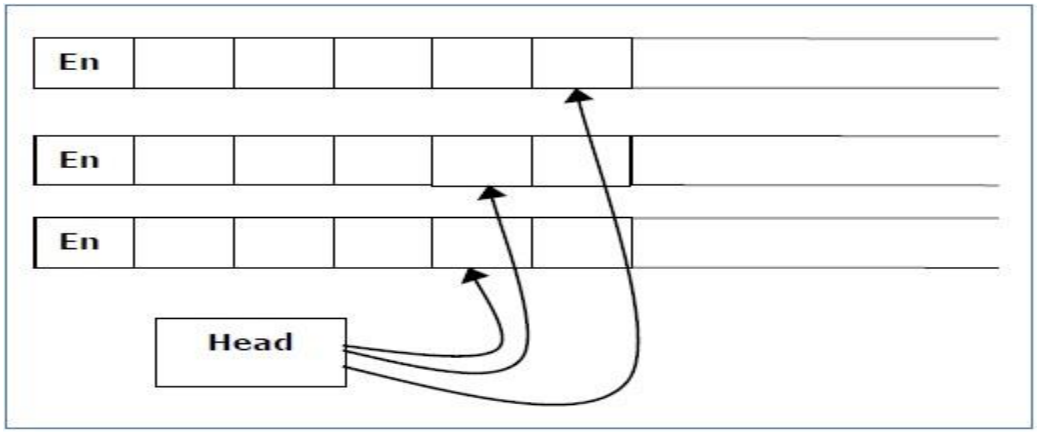




A. Multi-tape Turing Machine

Multi-tape Turing Machines have multiple tapes where each tape is accessed with a separate head. Each head can move independently of the other heads. Initially the input is on tape 1 and others are blank. At first, the first tape is occupied by the input and the other tapes are kept blank. Next, the machine reads consecutive symbols under its heads and the TM prints a symbol on each tape and moves its heads.





A Multi-tape Turing machine can be formally described as a 6-tuple (Q, X, B, δ, q_0, F) where –

- **Q** is a finite set of states
- **X** is the tape alphabet
- **B** is the blank symbol
- **δ** is a relation on states and symbols where

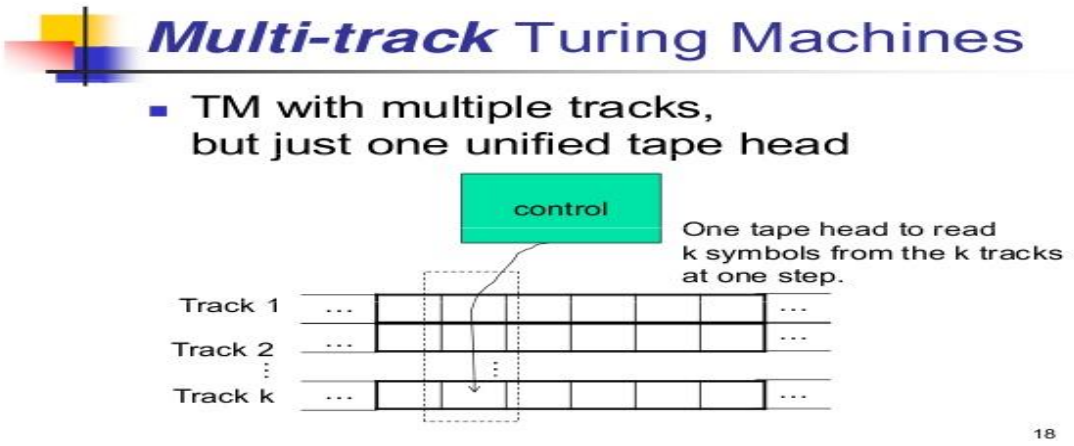
  $\delta: Q \times X^k \rightarrow Q \times (X \times \{Left\_shift, Right\_shift, No\_shift\})^k$

  where there is **k** number of tapes
- **q_0** is the initial state
- **F** is the set of final states

Note – Every Multi-tape Turing machine has an equivalent single-tape Turing machine.

B. Multi-track Turing Machine

Multi-track Turing machines, a specific type of Multi-tape Turing machine, contain multiple tracks but just one tape head reads and writes on all tracks. Here, a single tape head reads n symbols from **n** tracks at one step. It accepts recursively enumerable languages like a normal single-track single-tape Turing Machine accepts.



A Multi-track Turing machine can be formally described as a 6-tuple (Q, X, Σ, δ, q_0, F) where –

- **Q** is a finite set of states
- **X** is the tape alphabet
- **Σ** is the input alphabet
- **δ** is a relation on states and symbols where

  $\delta(Q_i, [a_1, a_2, a_3,....]) = (Q_j, [b_1, b_2, b_3,....], \text{Left\_shift or Right\_shift})$
- **q_0** is the initial state
- **F** is the set of final states

Note – For every single-track Turing Machine **S**, there is an equivalent multi-track Turing Machine **M** such that **L(S) = L(M)**.

C. Non-Deterministic Turing Machine

In a Non-Deterministic Turing Machine, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic. The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration.

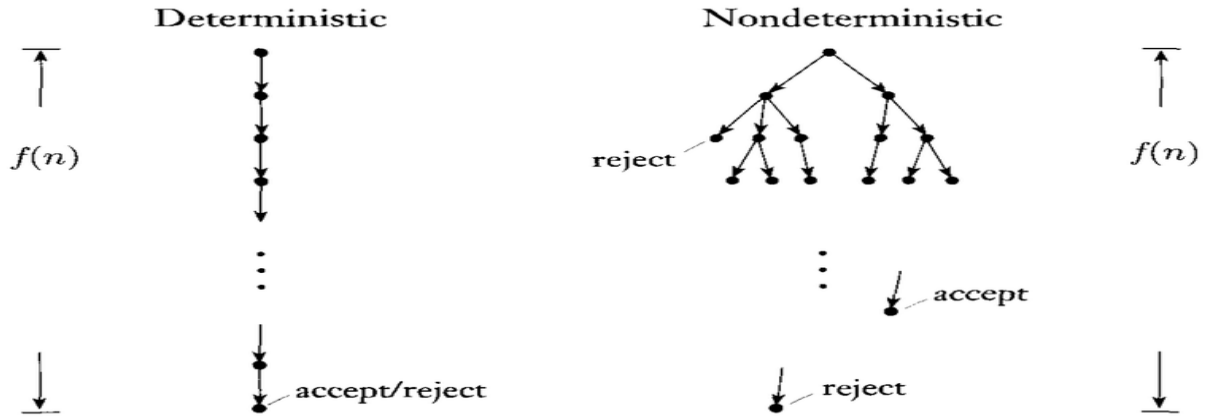
An input is accepted if there is at least one node of the tree which is an accept configuration, otherwise it is not accepted. If all branches of the computational tree halt on all inputs, the non-deterministic Turing Machine is called a **Decider** and if for some input, all branches are rejected, the input is also rejected.

A non-deterministic Turing machine can be formally defined as a 6-tuple (Q, X, Σ, δ, q_0, B, F) where –

- **Q** is a finite set of states

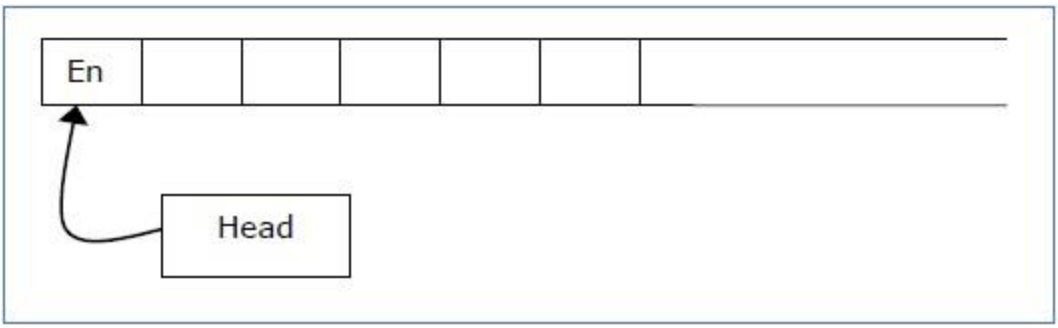
- **X** is the tape alphabet
- **Σ** is the input alphabet
- **δ** is a transition function;

  $\delta : Q \times X \rightarrow P(Q \times X \times \{Left\_shift, Right\_shift\})$.
- **q_0** is the initial state
- **B** is the blank symbol
- **F** is the set of final states



D. Semi-Infinite Tape Turing Machine

A Turing Machine with a semi-infinite tape has a left end but no right end. The left end is limited with an end marker.



It is a two-track tape –

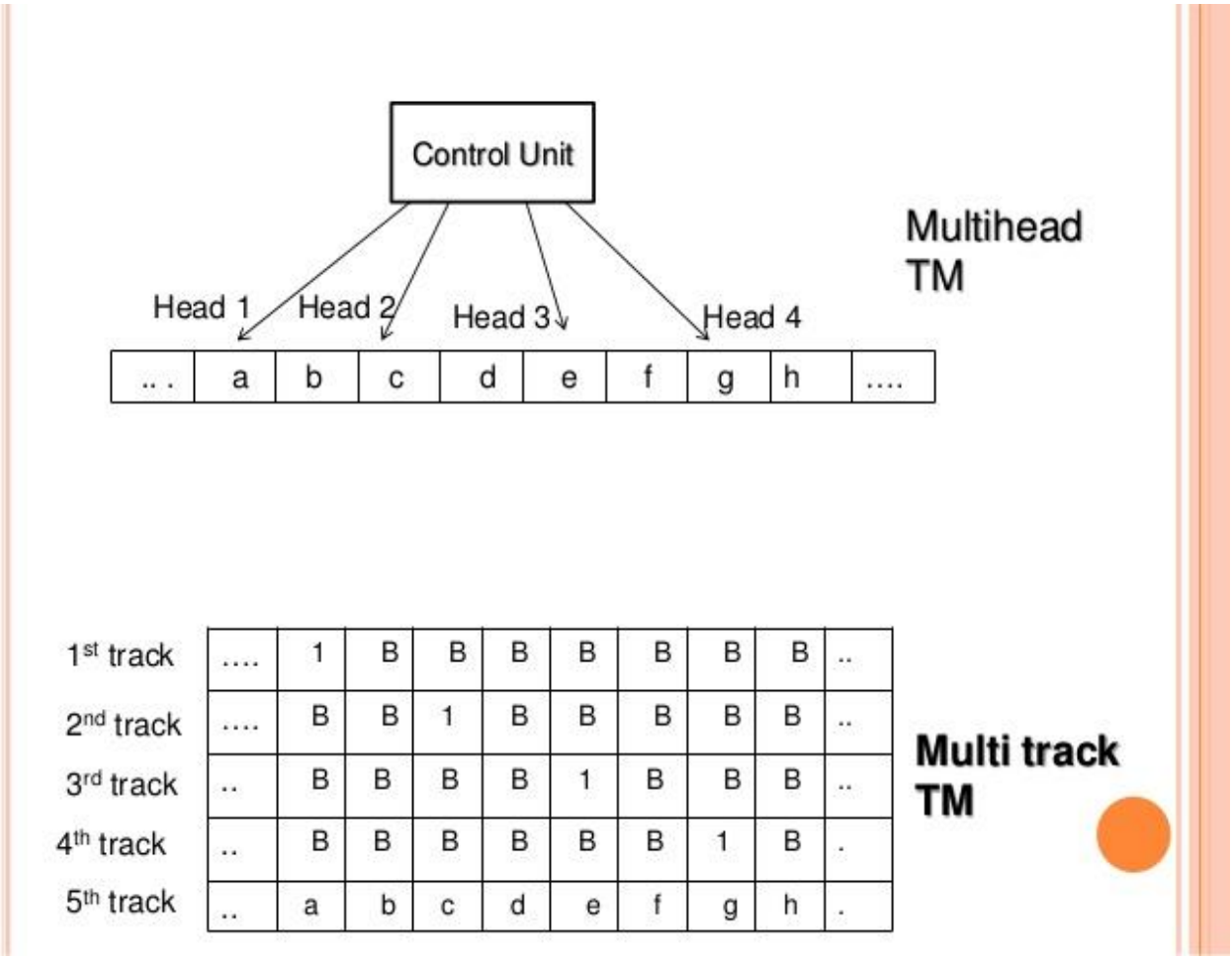
- **Upper track** – It represents the cells to the right of the initial head position.
- **Lower track** – It represents the cells to the left of the initial head position in reverse order.

The infinite length input string is initially written on the tape in contiguous tape cells.

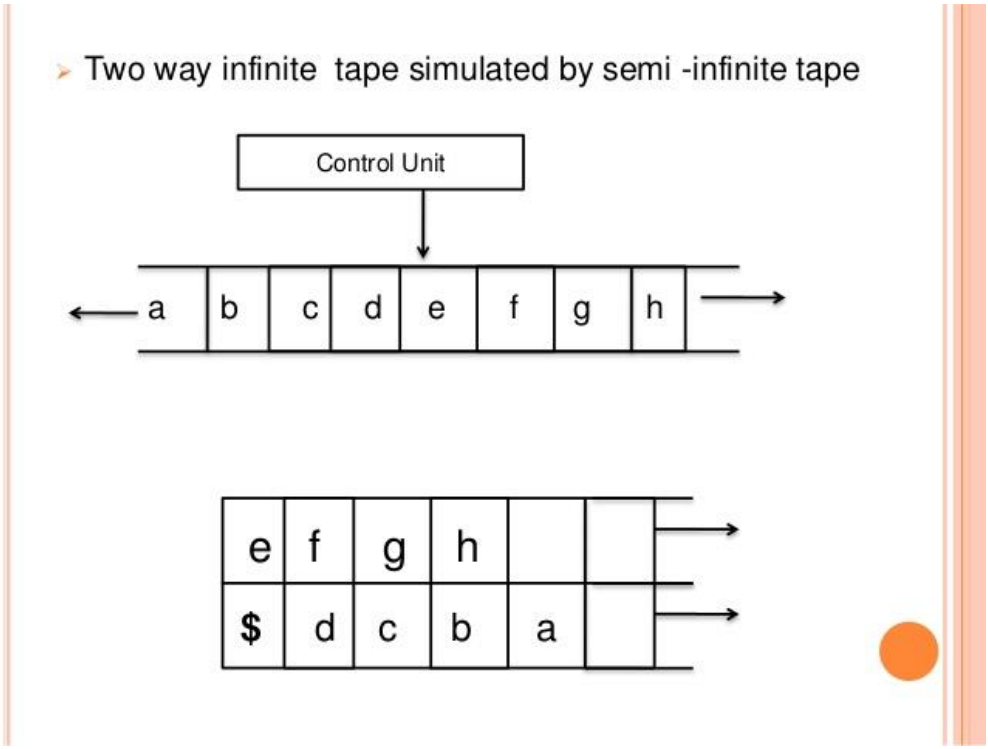
The machine starts from the initial state **q_0** and the head scans from the left end marker 'End'. In each step, it reads the symbol on the tape under its head. It writes a new symbol on that tape cell and then it moves the head either into left or right one tape cell. A transition function determines the actions to be taken.

It has two special states called **accept state** and **reject state**. If at any point of time it enters into the accepted state, the input is accepted and if it enters into the reject state, the input is rejected by the TM. In some cases, it continues to run infinitely without being accepted or rejected for some certain input symbols.

Note – Turing machines with semi-infinite tape are equivalent to standard Turing machines.



E. Two way Turing machine



A two way infinite tape TM can move indefinitely in either direction. This can be shown to be equivalent to a one way infinite tape TM by the following argument: It is clear that two one-way infinite tape TMs can be used to simulate a two-way infinite tape. If these are lined up ``side by side'' and the two tapes are interleaved, then a single one-way infinite TM can be constructed which is equivalent.

Variations on Turing Machines Before we proceed further with our exposition of Turing Machines as language acceptors, we will consider variations on the basic definition of Slide 10 and discuss, somewhat informally, how these extensions do not a↵ect the computing power of our Turing Machines. The variations however give us more flexibility when we discuss future aspects of Turing Machines. 2.6.1 Turing Machines with Two-Sided Infinite Tapes The first variation we consider is that of having a Turing Machine with a tape that is infinite on both sides as shown on Slides 45 and 46 (recall that in Def. 1, the tape was bounded on the left). Thus, conceptually, if the tape locations in Def. 1 could be enumerated as l0, l1, l2, ..., ln, ... we now can have the enumeration ..., ln, ..., l2, l1, l0, l1, l2, ..., ln, ... It turns out that such an additional feature does not increase the the computational power of our Turing Machine. This fact is stated as Theorem 30 on Slide 45. Unbounded Tapes on Both Sides Infinite on Both Sides ... l2 l1 l0 l1 l2 ... q Theorem 30. A Turing Machine with a double-sided infinite tape has equal computational power as a Turing Machine with a single-sided inifinite tape. Slide 45 Proof for Theorem 30. We are required to show that: 1. Any Turing Machine with a single-sided infnite tape can be simulated on a two-sided infinite tape Turing Machine. 2. Any two-sided infinite tape Turing Machine can be simulated on a Turing Machine with a single-sided infnite tape. Double-sided infinite tapes can express any Turing Machine with a single-side infinite tape by leaving the Turing Machine definition unaltered and simply not using the "negative index" side of the

tape. If the Turing Machine with a single-side infinite tape relies on hanging for its behaviour by moving one step after

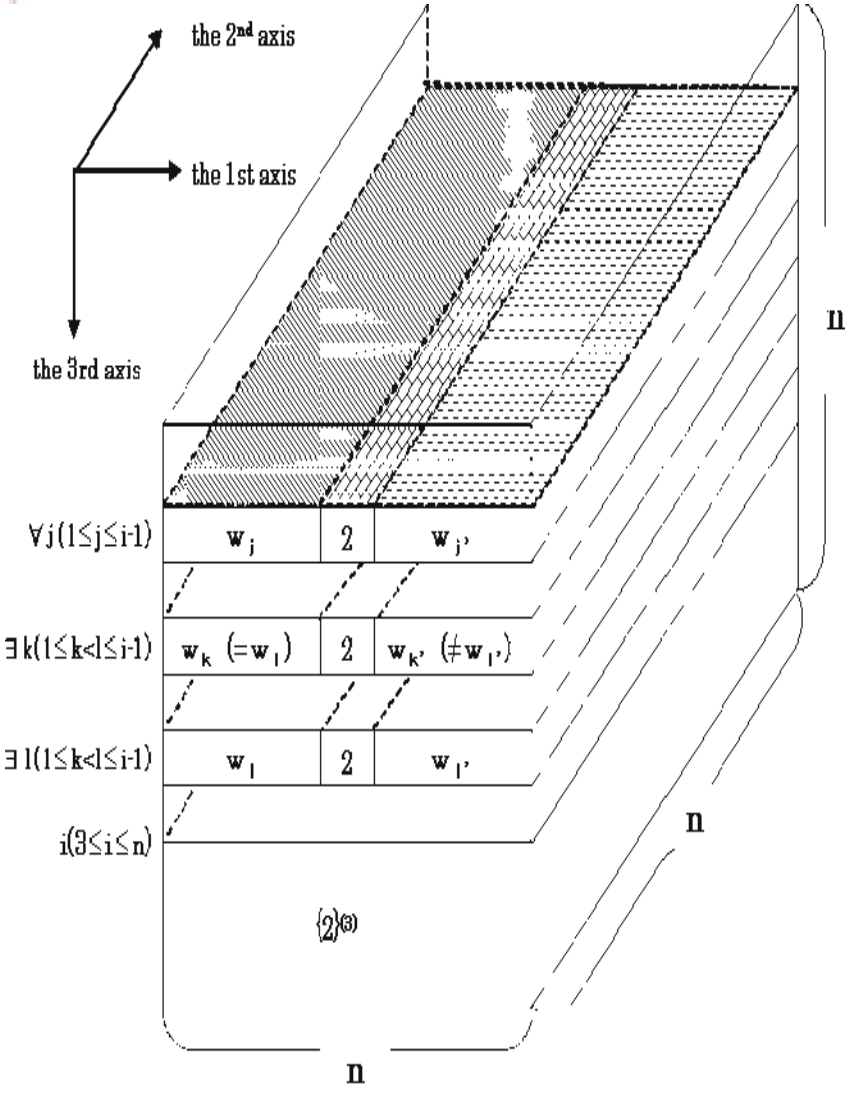
F. Multidimensional Turing Machine

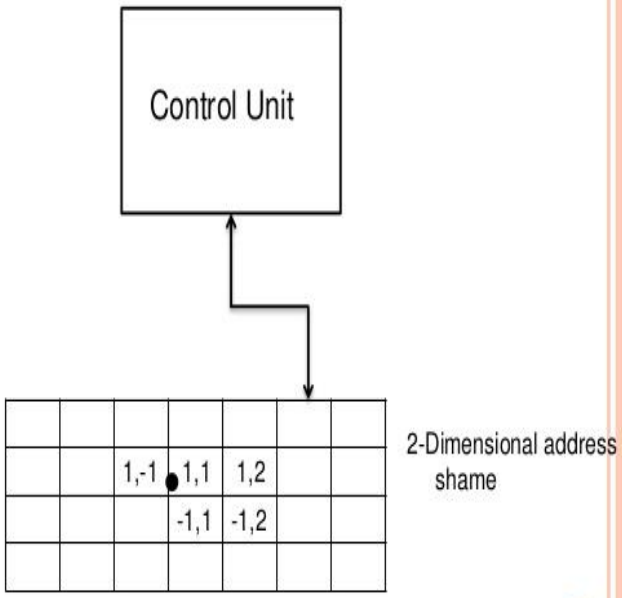
MULTIDIMENSIONAL TURING MACHINE

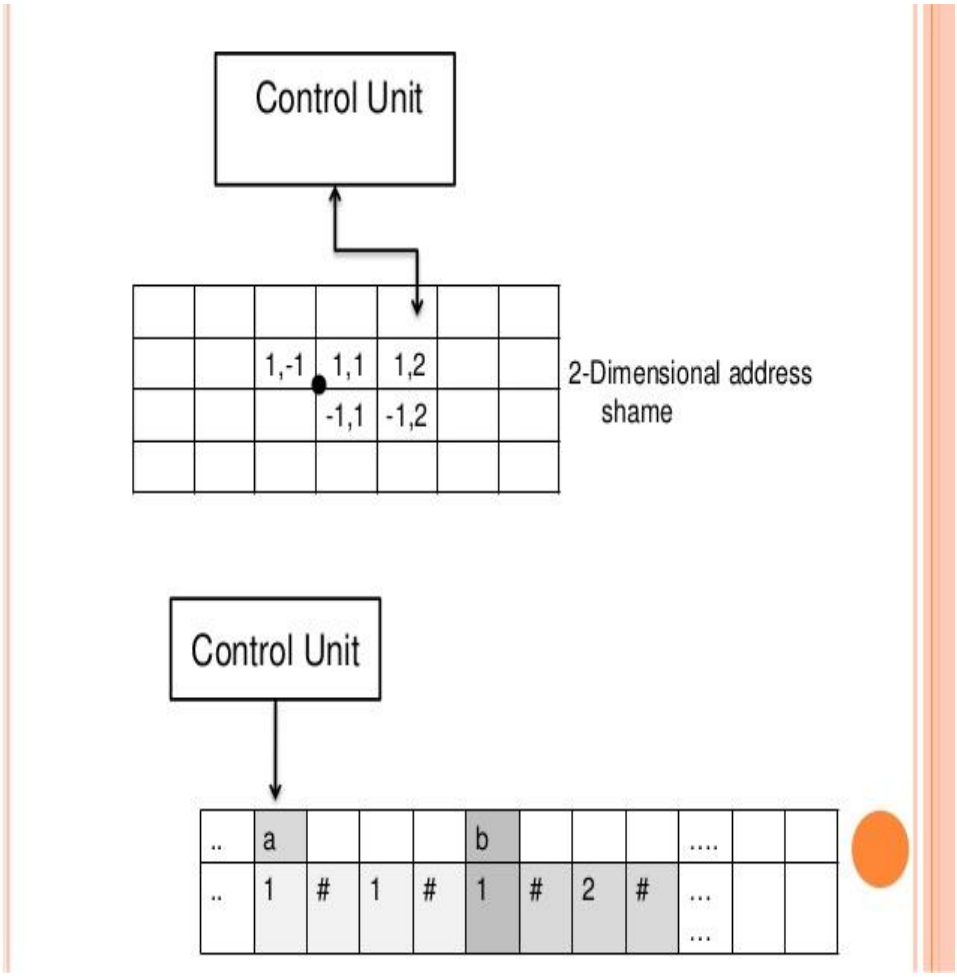
➢A Multidimensional TM has a multidimensional tape.
For example, a two-dimensional Turing machine would read and write on an infinite plane divided into squares, like a checkerboard.

➢ For a two- Dimensional Turing Machine transaction function define as:

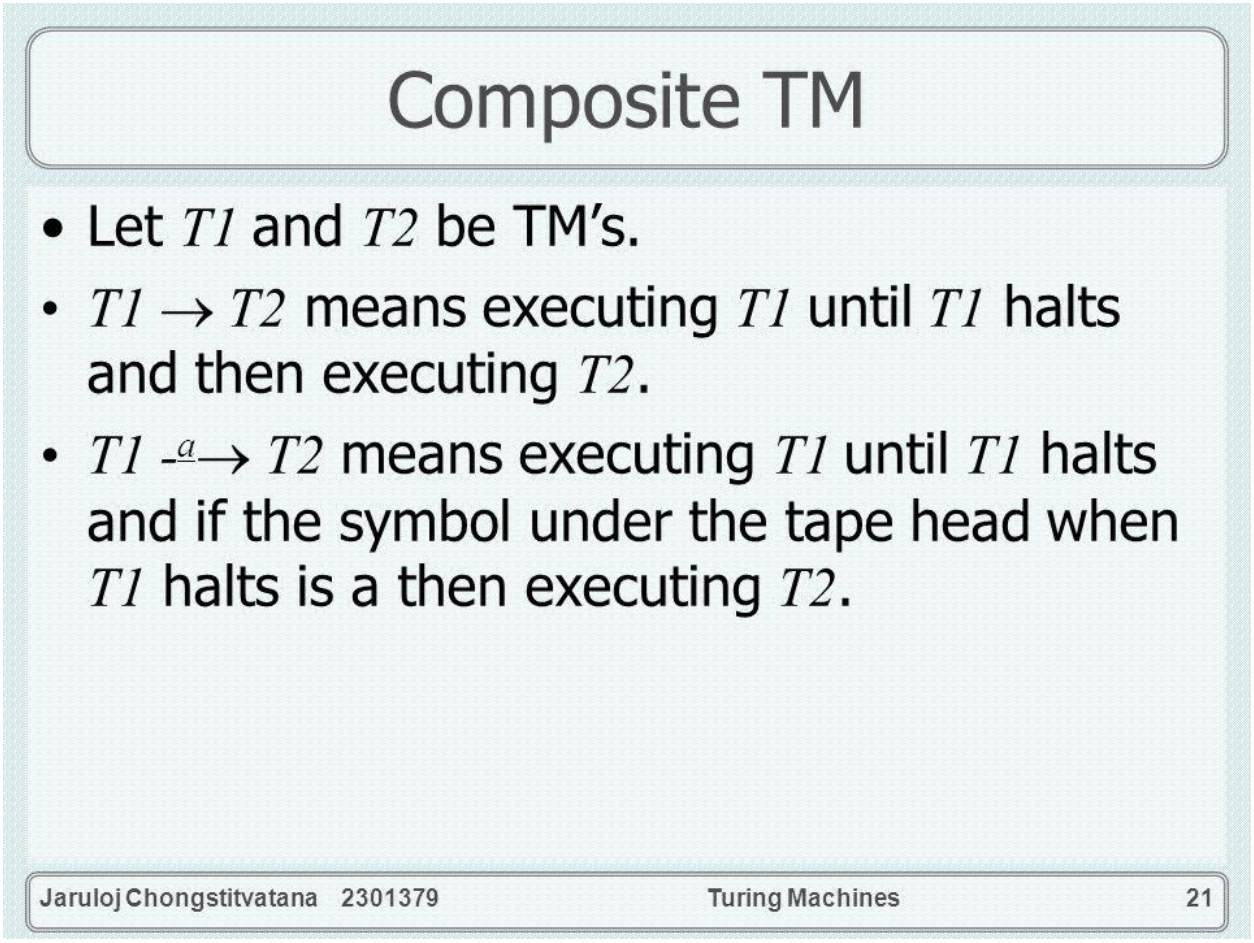
$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, U, D\}$$




Control Unit
1,-1
1,1
1,2
-1,1
-1,2
2-Dimensional address
shame



G. Composite Turing Machine



One may eliminate the returning of the read-write head to the beginning of the tape. Then the execution of the next Turing machine continues at the tape position where the previous one has finished its computation. In most of the cases this definition is more suitable, but then it would not behave as the usual function composition.

Using these composition rules, we may construct more complex Turing machines from simpler ones.

Some of the simpler Turing machines can be the following:

- moves the read-write head 1 position to the right
- observes 1 cell on the tape, the answer is "yes" or "no"
- moves the read-write head to the first or last position of the tape
- writes a fixed symbol to the tape
- copies the content of 1 cell to the next (left or right) cell
- ...

Composite Turing machines

- writes a symbol behind or before the input word
- deletes a symbol from the end or the beginning of the input word

- duplicates the input word
- mirrors the input word
- decide whether the length of the input word is even or odd
- compares the length of two input words
- exchanges representation between different number systems
- sums two words (= numbers)

Example

Let $\{0,1\}^*$ be the set of possible input words and let the following Turing machines be given:

1. $T_{\#?}$: observes the symbol under the read-write head; if it is a $\#$,then the answer is "yes" otherwise the answer is "no"

2. $T_{*?}$: observes the symbol under the read-write head; if it is a $*$,then the answer is "yes" otherwise the answer is "no"

3. $T_{1?}$: observes the symbol under the read-write head; if it is a 1,then the answer is "yes" otherwise the answer is "no"

4. $T_{\leftarrow}$: moves the read-write head one to the left

5. $T_{\rightarrow}$: moves the read-write head one to the right

6. T_0: writes a symbol 0 on the tape

7. T_1: writes a symbol 1 on the tape

8. $T_{\#}$: writes a symbol $\#$ on the tape

Applying composition operations, construct a Turing machine which duplicates the input word, i.e. it maps w to ww!

First we create a Turing machine for moving the read-write head to the rightmost position:

$$T_{\Rightarrow} = T_{*?} : \left((yes \rightarrow Stop) | (no \rightarrow T_{\rightarrow}) \right)^*$$.

The leftmost movement can be solved similarly:

$$T_{\Leftarrow} = T_{*?} : \left((yes \rightarrow Stop) | (no \rightarrow T_{\leftarrow}) \right)^*$$

The Turing machine which search for the first $\#$ to the right is the following:

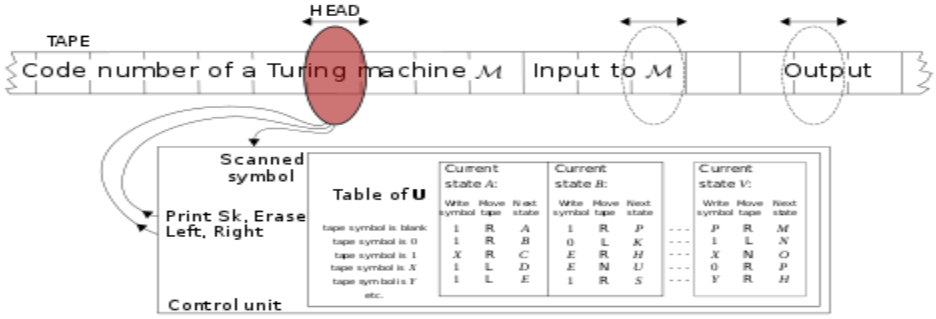
$$T_{\Rightarrow \#} = T_{\#?} : \left((yes \rightarrow Stop) | (no \rightarrow T_{\rightarrow}) \right)^*$$.

The Turing machine which search for the first $\#$ to the left is the following:

$$T_{\Leftarrow \#} = T_{\#?} : \left((yes \leftarrow Stop) | (no \rightarrow T_{\rightarrow}) \right)^*$$.

Universal Turing Machine

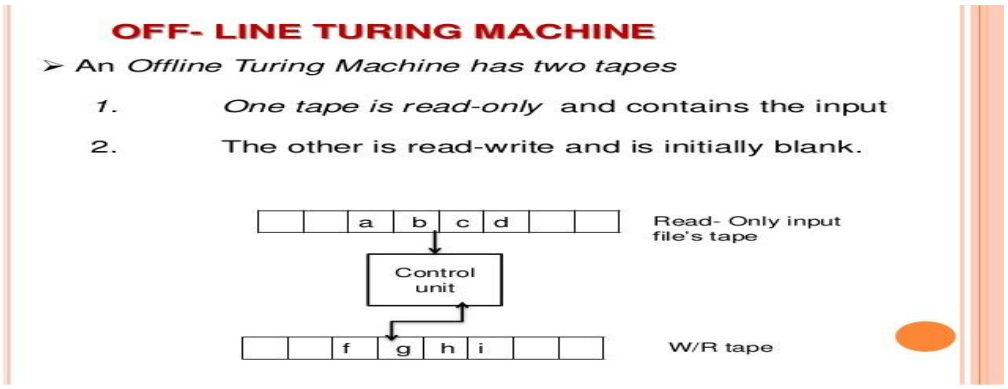
In computer science, a **universal Turing machine** (**UTM**) is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input. The universal machine essentially achieves this by reading both the description of the machine to be simulated as well as the input thereof from its own tape. Alan Turing introduced the idea of such a machine in 1936–1937. This principle is considered to be the origin of the idea of a stored-program computer used by John von Neumann in 1946 for the "Electronic Computing Instrument" that now bears von Neumann's name: the von Neumann architecture.[1]



Every Turing machine computes a certain fixed partial computable function from the input strings over its alphabet. In that sense it behaves like a computer with a fixed program. However, we can encode the action table of any Turing machine in a string. Thus we can construct a Turing machine that expects on its tape a string describing an action table followed by a string describing the input tape, and computes the tape that the encoded Turing machine would have computed. Turing described such a construction in complete detail in his 1936 paper:

> "It is possible to invent a single machine which can be used to compute any computable sequence. If this machine **U** is supplied with a tape on the beginning of which is written the S.D ["standard description" of an action table] of some computing machine **M**, then **U** will compute the same sequence as **M**." [3]

H. Offline TM

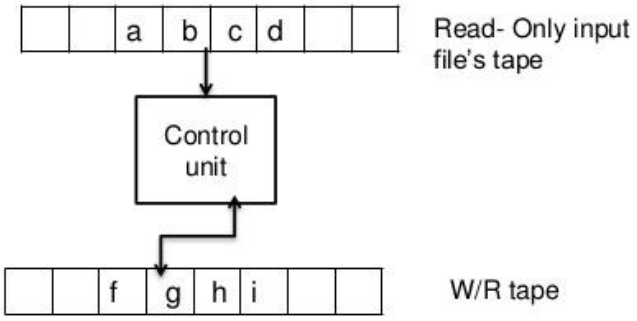


Simulation of off line TM

OFF- LINE TURING MACHINE

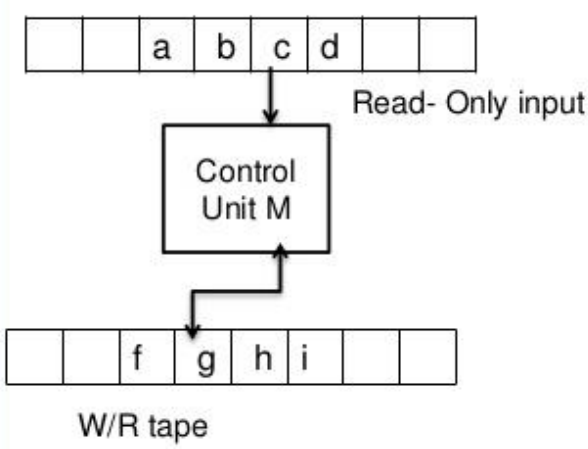
- An *Offline Turing Machine has two tapes*

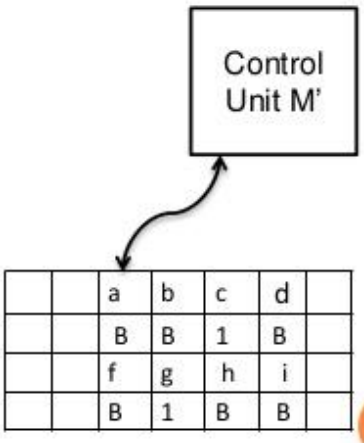
1. *One tape is read-only* and contains the input
2. The other is read-write and is initially blank.



SIMULATION

- A Standard TM simulated by Off-line TM
- An Off- line TM simulated by Standard TM



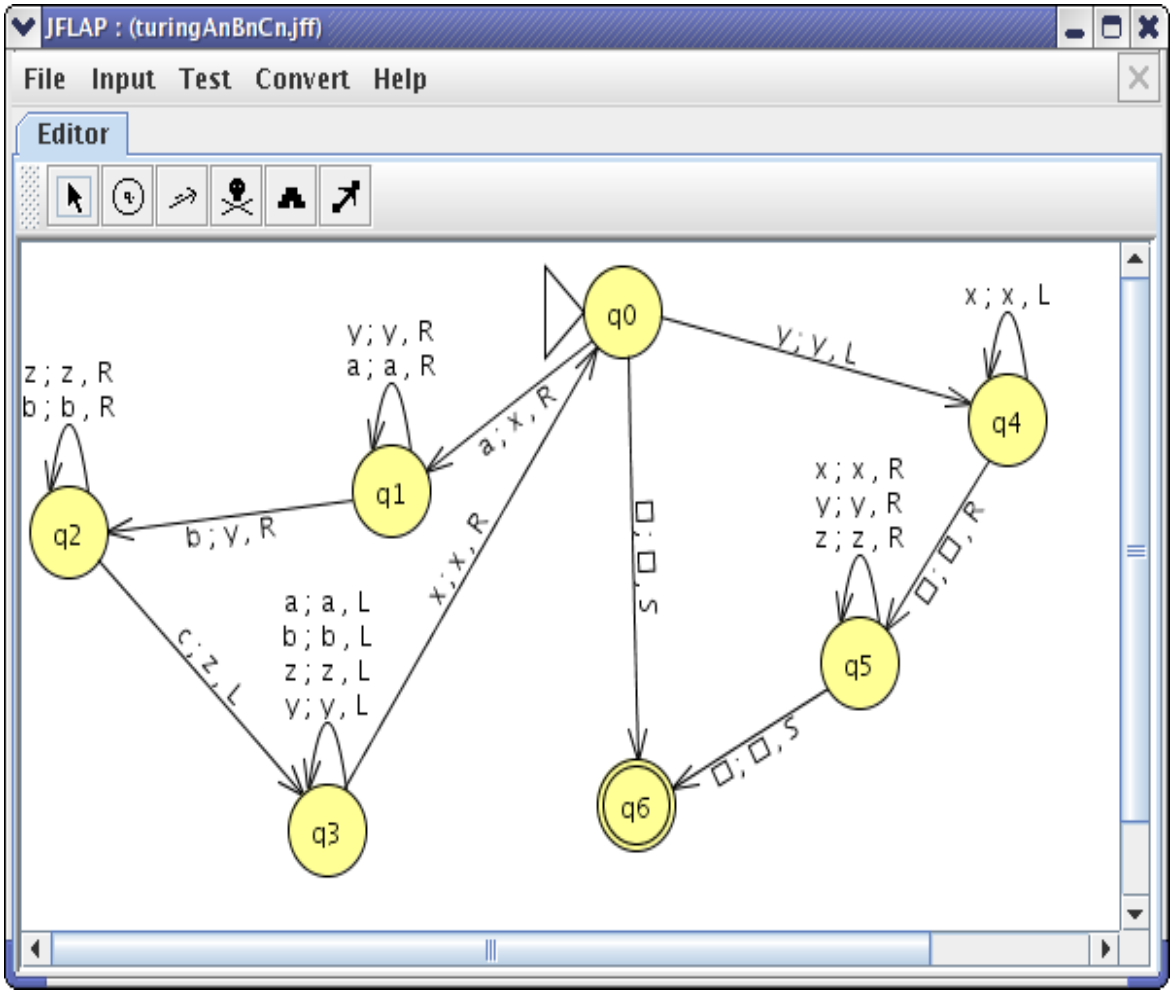


3. Design of TM

Turing Machine M as the septuple $M = (Q, \Sigma, \Gamma, \delta, q_s, \square, F)$ where

Q is the set of internal states $\{q_i \mid i \text{ is a nonnegative integer}\}$
Σ is the input alphabet
Γ is the finite set of symbols in the tape alphabet
δ is the transition function
S is $Q * \Gamma^n \rightarrow$ subset of $Q * \Gamma^n * \{L, S, R\}^n$
$\square$ is the blank symbol.
q_s (is member of Q) is the initial state
F (is a subset of Q) is the set of final states

Note that this definition includes both deterministic and nondeterministic Turing machines.



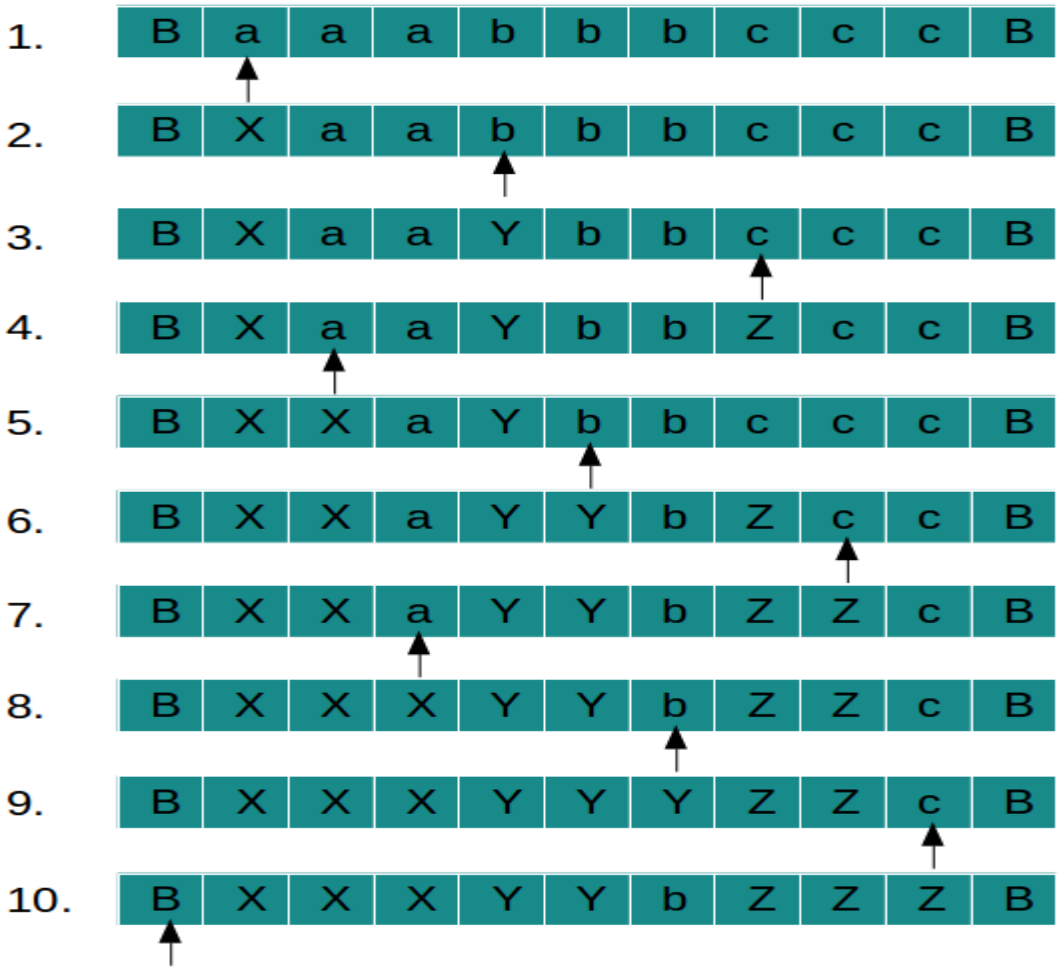
Turing machine for $a^nb^nc^n$ | n ≥ 1

Previously we have seen example of turing machine for a^nb^n | n ≥ 1 We will use the same concept for $a^nb^nc^n$ | n ≥ 1 also.

Approach for $a^nb^nc^n$ | n ≥ 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.

TAPE movement for string "aaabbbccc":



Explanation of TAPE movement

Step 1-2

1. Input is given as "aaabbbccc" (scan string from left to right)
2. Mark 'a' as 'X' and move one step right
3. Reach unmarked 'b' and pass every 'a' and 'Y'(in case) on the way to 'b'

Step 3-4

4. Mark 'b' as 'Y' and move one step right
5. Reach unmarked 'c' and pass every 'b' and 'Z'(in case) on the way to 'c'
6. Mark 'c' as 'Z' and move one step left cause now we have to repeat process
7. Reach unmarked 'X' and pass every 'Z', 'b', 'Y', 'a' on the way to 'X'
8. Move to 'a' and repeat the process

Step 5-9

9. Step 1-4 are repeated

Step 10

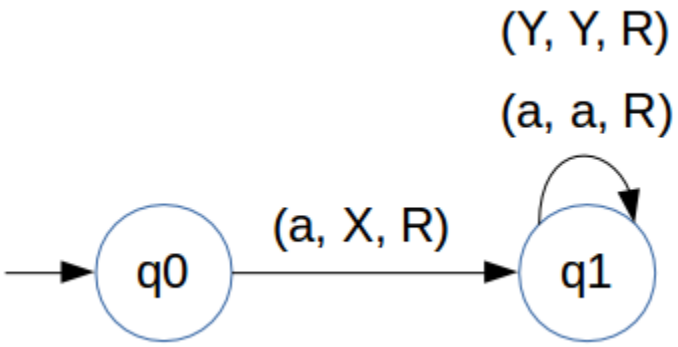
10. When TAPE header reaches 'X' and next symbol is 'Y' that means 'a's are finished
11. Check for 'b's and 'c's by going till BLANK(in right) and passing 'Y' and 'Z' on the way
12. If no 'b' and 'c' are not found that means string is accepted

---

State Transition Diagram

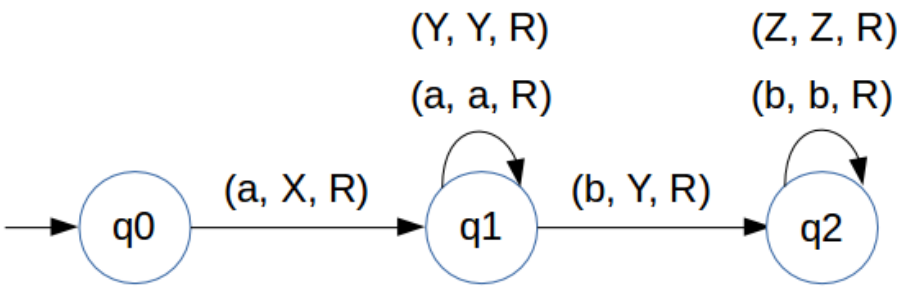
We have designed state transition diagram for $a^n b^n c^n \mid n \geq 1$ as follows:

1. Following Steps:
   a. Mark 'a' with 'X' and move towards unmarked 'b'
   b. Move towards unmarked 'b' by passing all 'a's
   c. To move towards unmarked 'b' also pass all 'Y's if exist

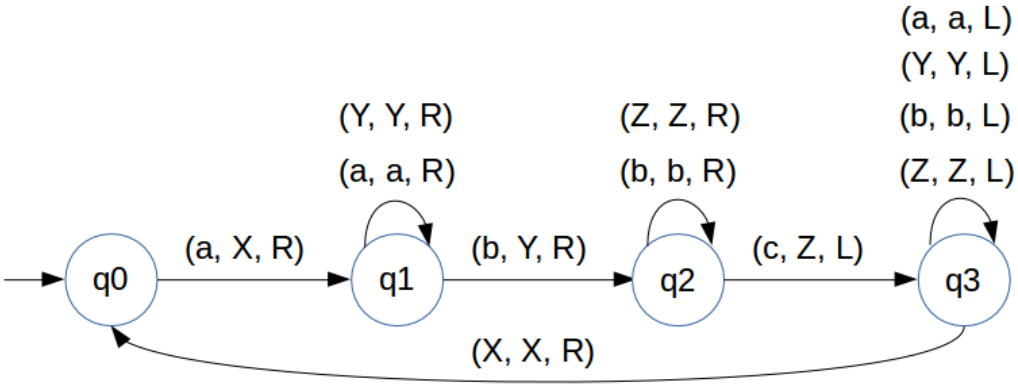


2. Following Steps:
   a. Mark 'b' with 'Y' and move towards unmarked 'c'

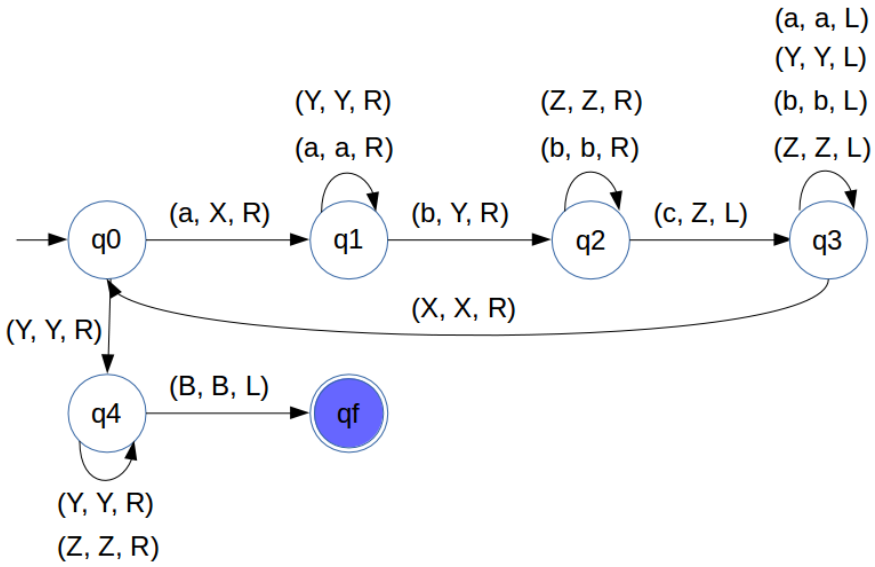
b. Move towards unmarked 'c' by passing all 'b's
c. To move towards unmarked 'c' also pass all 'Z's if exist



3. Following Steps:
   a. Mark 'c' with 'Z' and move towards first 'X' (in left)
   b. Move towards first 'X' by passing all 'Z's, 'b's, 'Y's and 'a's
   c. When 'X' is reached just move one step right by doing nothing.



4. To check all the 'a's, 'b's and 'c's are over add loops for checking 'Y' and 'Z' after "we get 'X' followed by 'Y'"
   To reach final state(qf) just replace BLANK with BLANK and move either direction



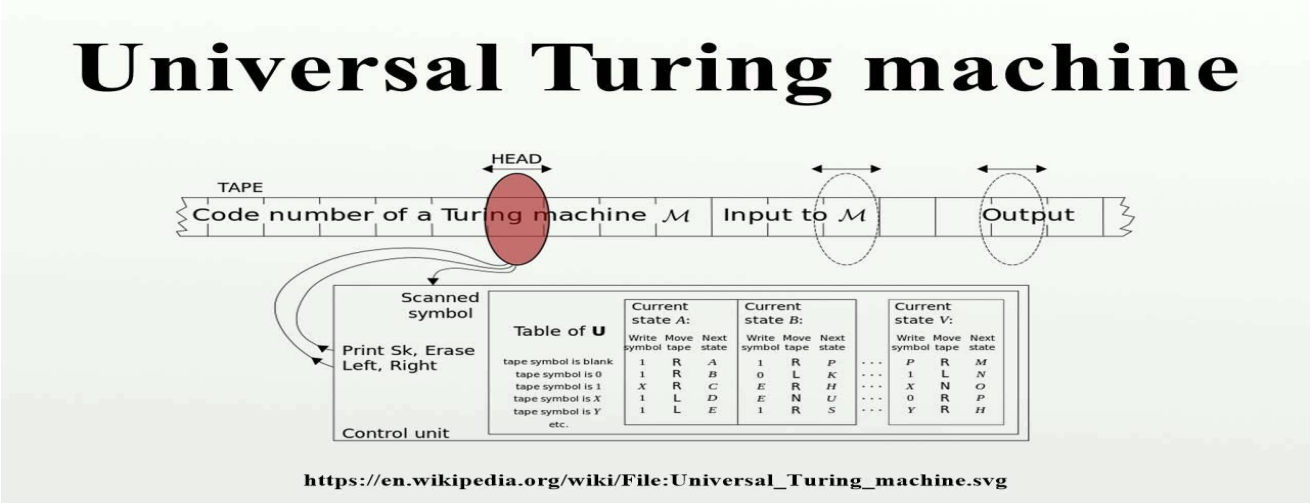
4. Universal Turing Machine

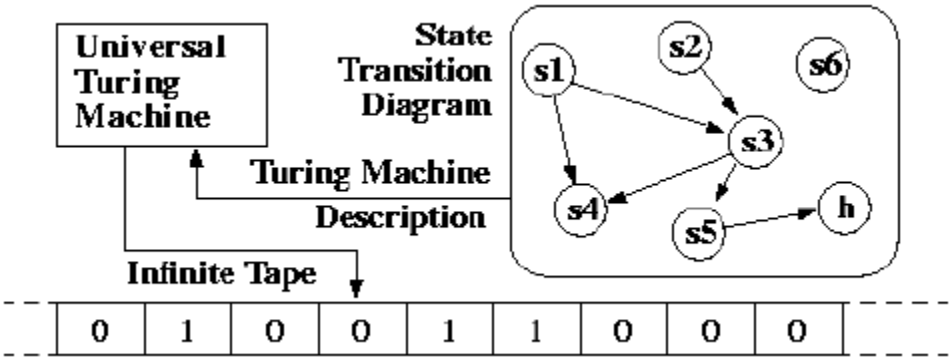
A "Universal Turing Machine," or 'UTM,' can emulate any other specific Turing machine, by defining states and symbols. The UTM is defined with certain capabilities.

- The UTM can define the symbols that the specific Turing machine will use.
- It can define the symbols that encode the states and transition rules for the specific Turing machine.
- It can encode the rules for that specific Turing machine onto the input tape.

A single-tape UTM needs to define a marker to mark the end of the "specific" program and the start of the specific machine's initial tape. It must also shuffle the read/write head between the specific TM's program and its data. As noted, it is simpler to describe a UTM with multiple tapes.

The Universal Turing Machine is remarkably similar to the Von Neumann model of a computer, where both programs and data can be stored on the same medium. Any modern computer capable of copying a program file from one medium to another, and later running that program, follows this architecture.





The universal Turing machine

A **universal Turing machine**, a machine that can do any computation if the appropriate program is provided, was the first description of a modern computer. It can be proved that a very powerful computer and a universal Turing machine can compute the same thing. We need only provide the data and the program—the description of how to do the computation—to either machine. In fact, a universal Turing machine is capable of computing anything that is computable.

5. Computable function

Computable functions are the basic objects of study in computability theory. Computable functions are the formalized analogue of the intuitive notion of algorithm, in the sense that a function is computable if there exists an algorithm that can do the job of the function, i.e. given an input of the function domain it can return the corresponding output. Computable functions are used to discuss computability without referring to any concrete model of computation such as Turing machines or register machines. Any definition, however, must make reference to some specific model of computation but all valid definitions yield the same class of functions. Particular models of computability that give rise to the set of computable functions are the Turing-computable functions and the μ-recursive functions.

Before the precise definition of computable function, mathematicians often used the informal term *effectively calculable*. This term has since come to be identified with the computable functions. Note that the effective computability of these functions does not imply that they can be *efficiently* computed (i.e. computed within a reasonable amount of time). In fact, for some effectively calculable functions it can be shown that any algorithm that computes them will be very inefficient in the sense that the running time of the algorithm increases exponentially (or even superexponentially) with the length of the input. The fields of feasible computability and computational complexity study functions that can be computed efficiently.

According to the Church–Turing thesis, computable functions are exactly the functions that can be calculated using a mechanical calculation device given unlimited amounts of time and storage space. Equivalently, this thesis states that a function is computable if and only if it has an algorithm. Note that an algorithm in this sense is understood to be a sequence of steps a person with unlimited time and an unlimited supply of pen and paper could follow.

The Blum axioms can be used to define an abstract computational complexity theory on the set of computable functions. In computational complexity theory, the problem of determining the complexity of a computable function is known as a function problem.

As counterparts to this informal description, there exist multiple formal, mathematical definitions. The class of computable functions can be defined in many equivalent models of computation, including

- **Turing machines**
- **μ-recursive functions**
- **Lambda calculus**
- **Post machines** (**Post–Turing machines** and **tag machines**).
- **Register machines**

Although these models use different representations for the functions, their inputs and their outputs, translations exist between any two models, and so every model describes essentially the same class of functions, giving rise to the opinion that formal computability is both natural and not too narrow.[2]

For example, one can formalize computable functions as μ-recursive functions, which are partial functions that take finite tuples of natural numbers and return a single natural number (just as above). They are the smallest class of partial functions that includes the constant, successor, and projection functions, and is closed under composition, primitive recursion, and the μ operator.

Equivalently, computable functions can be formalized as functions which can be calculated by an idealized computing agent such as a Turing machine or a register machine. Formally speaking, a partial function can be calculated if and only if there exists a computer program with the following properties:

1. If is defined, then the program will terminate on the input with the value stored in the computer memory.
2. If is undefined, then the program never terminates on the input .

6. Recursive enumerable language

RECURSIVE AND RECURSIVELY ENUMERABLE LANGUAGE

The Turing machine may

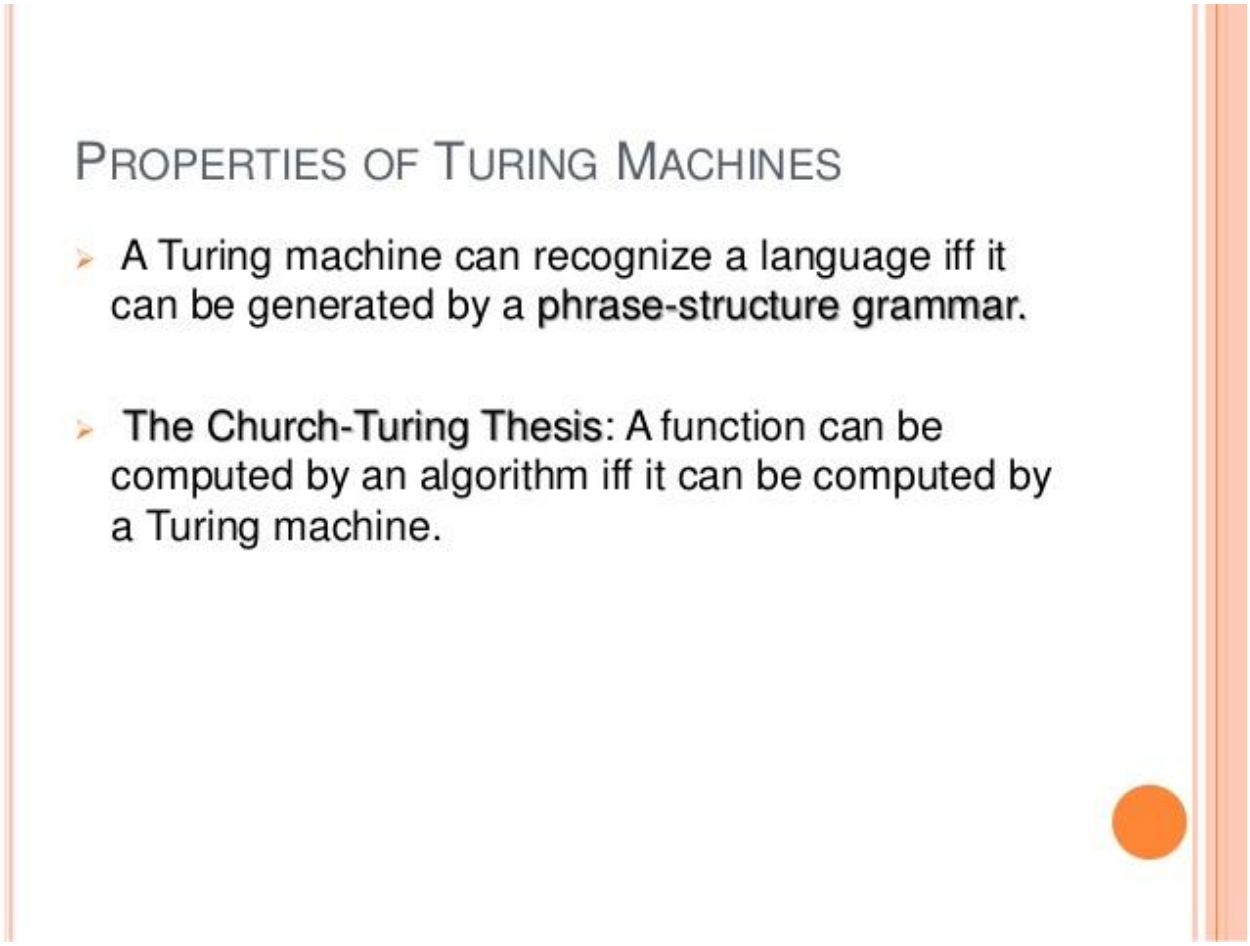
1. Halt and accept the input
2. Halt and reject the input, or
3. Never halt /loop.

Recursively Enumerable Language:
There is a TM for a language which accept every string otherwise not..

Recursive Language:
There is a TM for a language which halt on every string.

UNIVERSAL LANGUAGE AND TURING MACHINE

- The universal language **Lu** is the set of binary strings that encode a pair (M , w) where w is accepted by M
- A Universal Turing machine (**UTM**) is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input.



7. Linear bounded automata

A linear bounded automaton is a multi-track non-deterministic Turing machine with a tape of some bounded finite length.

Length = function (Length of the initial input string, constant c)

Here,

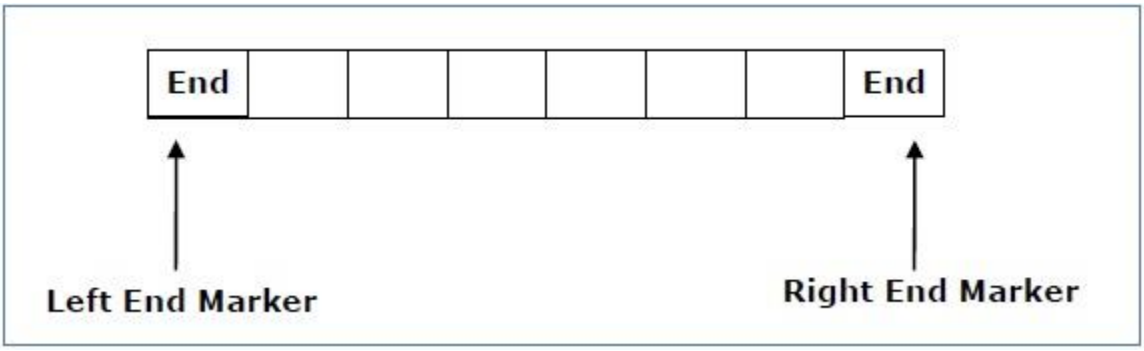
Memory information ≤ c × Input information

The computation is restricted to the constant bounded area. The input alphabet contains two special symbols which serve as left end markers and right end markers which mean the transitions neither move to the left of the left end marker nor to the right of the right end marker of the tape.

A linear bounded automaton can be defined as an 8-tuple (Q, X, Σ, q_0, ML, MR, δ, F) where –

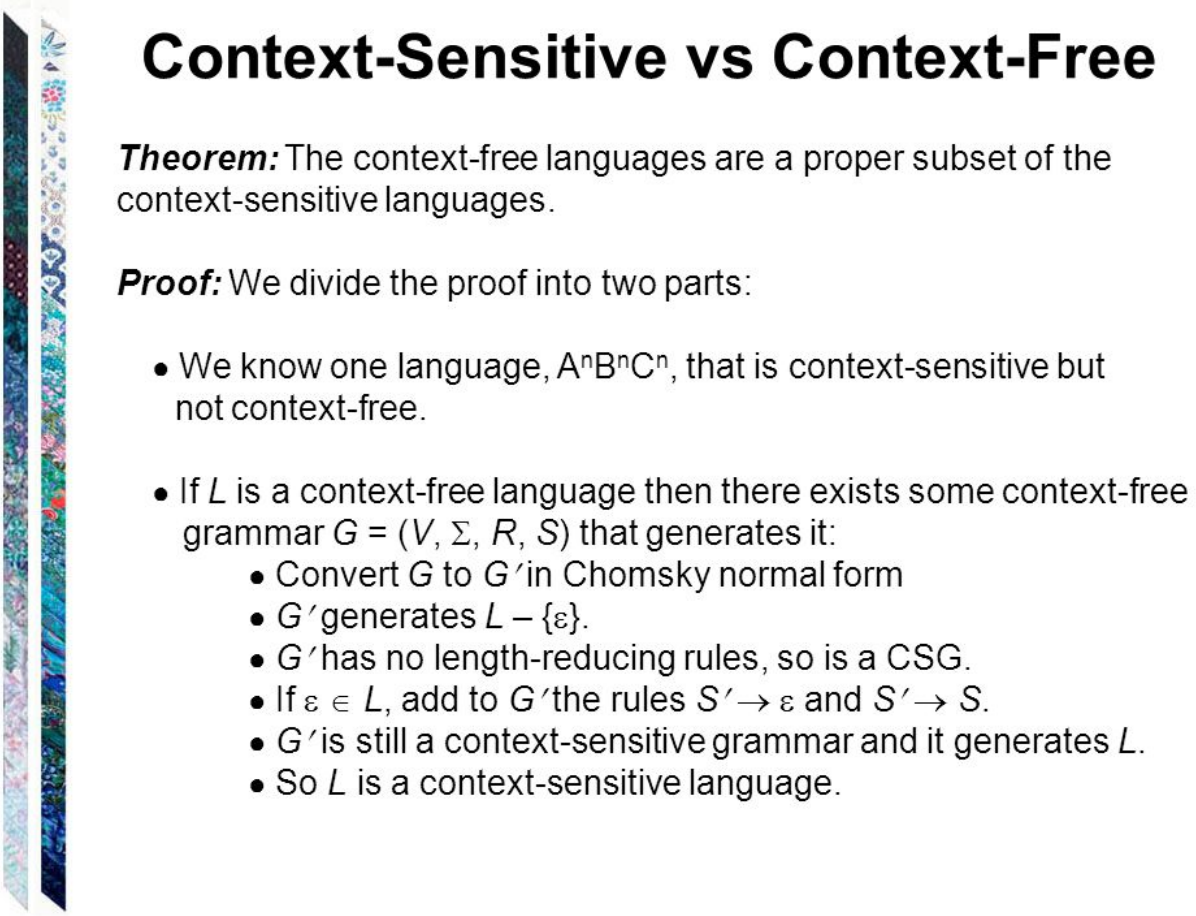
- **Q** is a finite set of states
- **X** is the tape alphabet
- **Σ** is the input alphabet
- **q_0** is the initial state
- **M_L** is the left end marker

- **M_R** is the right end marker where $M_R \neq M_L$
- **δ** is a transition function which maps each pair (state, tape symbol) to (state, tape symbol, Constant 'c') where c can be 0 or +1 or -1
- **F** is the set of final states



A deterministic linear bounded automaton is always **context-sensitive** and the linear bounded automaton with empty language is **undecidable.**.

8. Context sensitive language



THE FOLLOWING PAGES CONTAIN COMPONENTS OF TABLE 1.

Language	*Grammar*	*Automaton*	*Example*
Recursively enumerable	Unrestricted	Turing machine	Any computable function
Context-sensitive	Context-sensitive	Linear-bounded automaton	$a^n b^n c^n$
Context-free	Context-free	Non-deterministic pushdown automaton	$a^n b^n$
Regular	Regular	Finite-state automaton	a^n

Table 1
The Classical Chomsky Hierarchy (Chomsky, 1959): Each row of the table represer one of four language complexity levels, starting from the (most restrictive) regul languages to the (most general) recursively enumerable languages.

Context-Sensitive Grammars and Languages

Example of a grammar that is not context-sensitive:

$S \rightarrow aSb$
$S \rightarrow \varepsilon$

An equivalent, context-sensitive grammar:

$S \rightarrow \varepsilon$
$S \rightarrow T$
$T \rightarrow aTb$
$T \rightarrow ab$

Examples: Context Sensitive Grammars

- Multiple non-terminal on the left
  - P = {S → aSBc | abc, cB → Bc, bB → bb}
  - Generates the language $a^n b^n c^n$, n>=1
  - Exemplar Process:

S => aSBc => aabcBc => aabBcc => aabbcc

How to generate $a^n b^n c^n d^n$, n>=1?

P = {S → aBSCd | abcd, Ba → aB, dC →Cd, cC → cc, Bb → bb }

Context-Sensitive Languages (Type 1)

- Example language

  L = { $a^n b^n c^n$ | n >= 1}

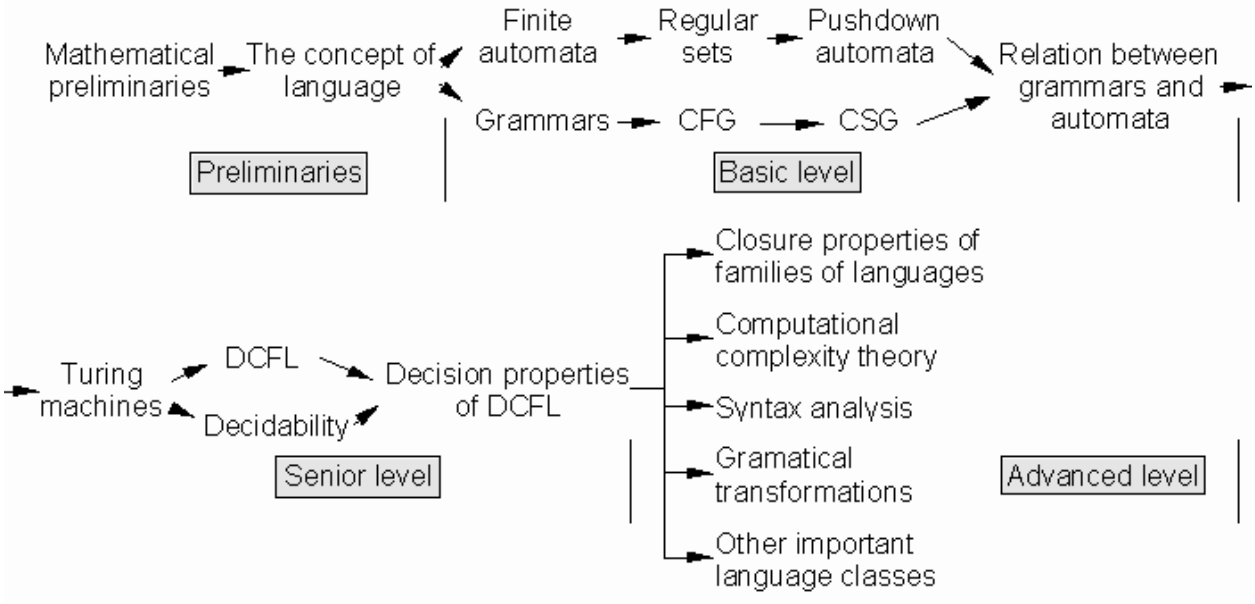
- More powerful than context-free grammars
- All context-free languages are also context-sensitive
- Not all context-sensitive languages are context-free

- **REL (Recursively Enumerable Language)**
- **CSL (Context Sensitive Language)**
- **CFL (Context Free Language)**
- **RL (Regular Language)**

Examples of Formal Language

- **simple matching language :** $L_m = \{a^n b^n \mid n \geq 0\}$ **CFL**
- **double matching language :** $L_{dm} = \{a^n b^n c^n \mid n \geq 0\}$ **CSL**
- **mirror image language :** $L_{mi} = \{\omega\omega^R \mid \omega \in V_T^*\}$ **CFL**
- **palindrome language :** $L_r = \{\omega \mid \omega = \omega^R\}$ **CFL**
- **parenthesis language :** $L_p = \{\omega \mid \omega$: balenced parenthesis$\}$ **CFL**

MMLab, University of Seoul



9. Counter machine

A **counter machine** is an abstract machine used in formal logic and theoretical computer science to model computation. It is the most primitive of the four types of register machines. A counter machine comprises a set of one or more unbounded *registers*, each of which can hold a single non-negative integer, and a list of (usually sequential) arithmetic and control instructions for the machine to follow. The counter machine is typically used in the process of designing parallel

algorithms in relation to the mutual exclusion principle. When used in this manner, the counter machine is used to model the discrete time-steps of a computational system in relation to memory accesses. By modeling computations in relation to the memory accesses for each respective computational step, parallel algorithms may be designed in such a matter to avoid interlocking, the simultaneous writing operation by two (or more) threads to the same memory address.

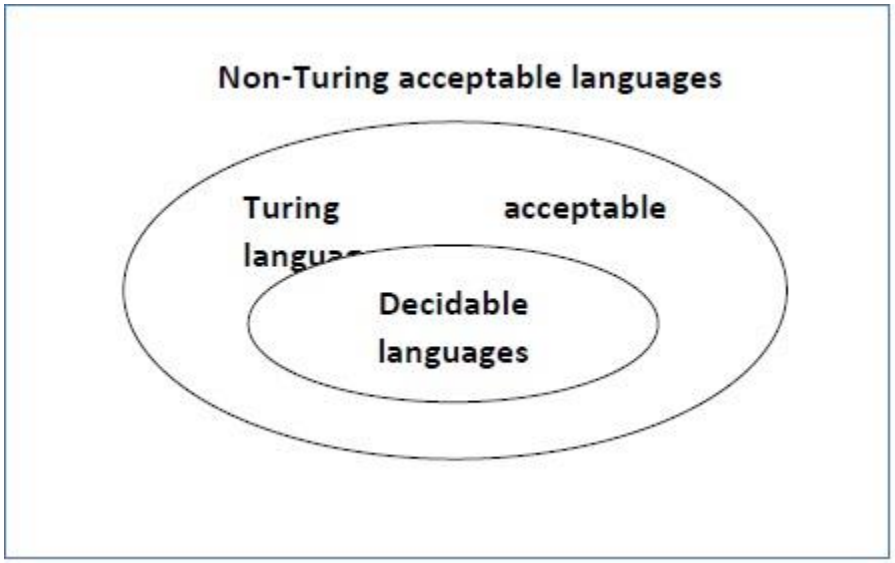
For a given counter machine model the instruction set is tiny—from just one to six or seven instructions. Most models contain a few arithmetic operations and at least one conditional operation (if *condition* is true, then jump). Three *base models*, each using three instructions, are drawn from the following collection. (The abbreviations are arbitrary.)

- CLR (r): CLeaR register *r*. (Set *r* to zero.)
- INC (r): INCrement the contents of register *r*.
- DEC (r): DECrement the contents of register *r*.
- CPY (r_j, r_k): CoPY the contents of register r_j to register r_k leaving the contents of r_j intact.
- JZ (r, z): IF register *r* contains Zero THEN Jump to instruction *z* ELSE continue in sequence.
- JE (r_j, r_k, z): IF the contents of register r_j Equals the contents of register r_k THEN Jump to instruction *z* ELSE continue in sequence.

UNIT V:

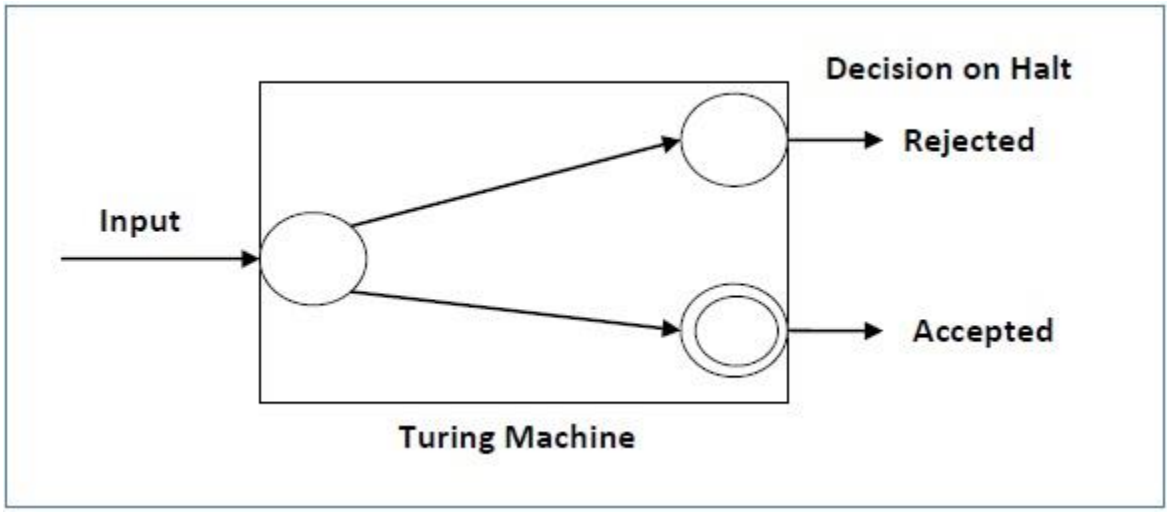
1. Decidability and Undecidability of problems

A language is called **Decidable** or **Recursive** if there is a Turing machine which accepts and halts on every input string **w**. Every decidable language is Turing-Acceptable.



A decision problem **P** is decidable if the language **L** of all yes instances to **P** is decidable.

For a decidable language, for each input string, the TM halts either at the accept or the reject state as depicted in the following diagram –



Example 1

Find out whether the following problem is decidable or not –

Is a number 'm' prime?

Solution

Prime numbers = {2, 3, 5, 7, 11, 13, ……..}

Divide the number **'m'** by all the numbers between '2' and '√m' starting from '2'.

If any of these numbers produce a remainder zero, then it goes to the "Rejected state", otherwise it goes to the "Accepted state". So, here the answer could be made by 'Yes' or 'No'.

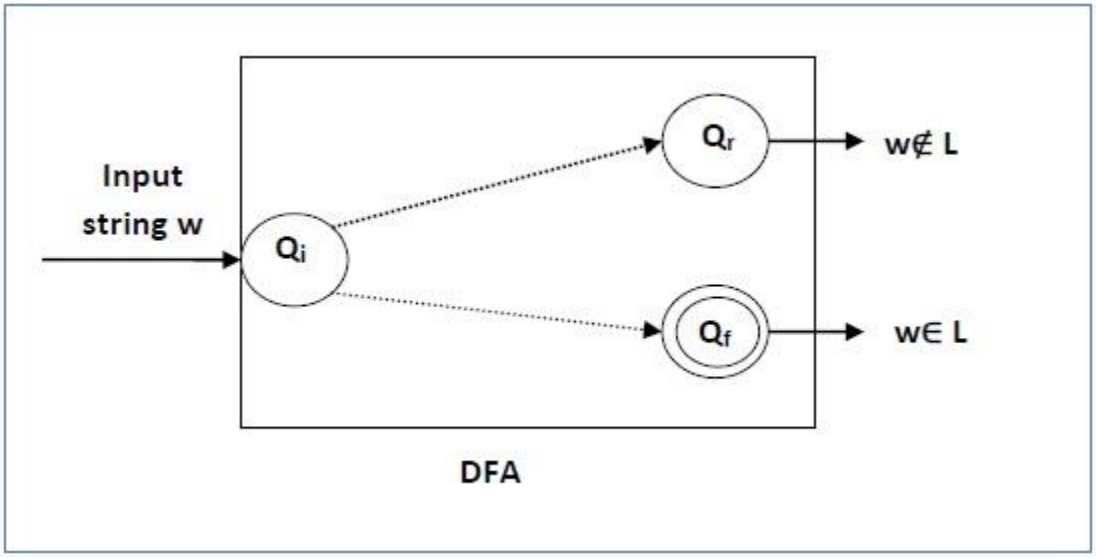
Hence, it is a decidable problem.

Example 2

Given a regular language **L** and string **w**, how can we check if **w ∈ L**?

Solution

Take the DFA that accepts **L** and check if **w** is accepted



Some more decidable problems are –

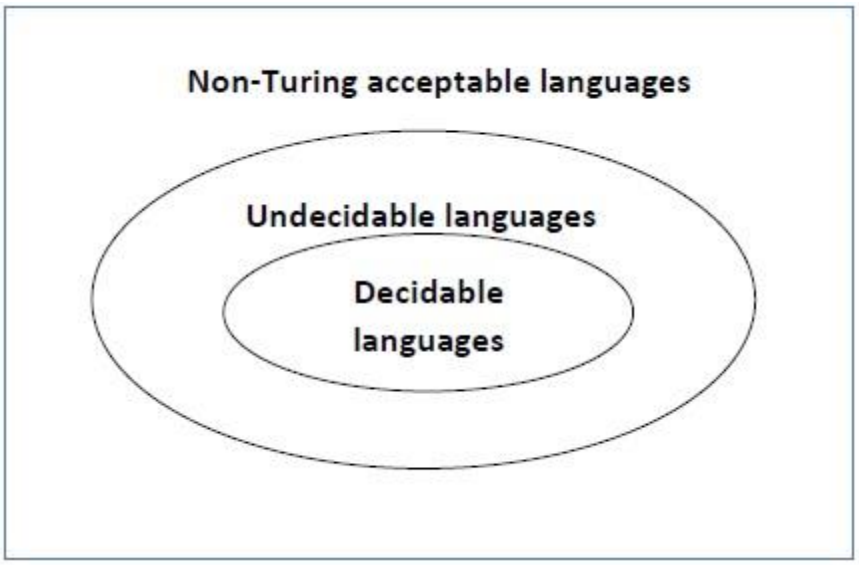
- Does DFA accept the empty language?
- Is $L_1 \cap L_2 = \emptyset$ for regular sets?

Note –

- If a language **L** is decidable, then its complement **L'** is also decidable
- If a language is decidable, then there is an enumerator for it.

For an undecidable language, there is no Turing Machine which accepts the language and makes a decision for every input string **w** (TM can make decision for some input string though). A decision problem **P** is called "undecidable" if the

language **L** of all yes instances to **P** is not decidable. Undecidable languages are not recursive languages, but sometimes, they may be recursively enumerable languages.



Example

- The halting problem of Turing machine
- The mortality problem
- The mortal matrix problem
- The Post correspondence problem, etc.

2. Properties of recursive & recursively enumerable languages

In mathematics, logic and computer science, a formal language is called **recursively enumerable** (also **recognizable**, **partially decidable**, **semidecidable**, **Turing-acceptable** or **Turing-recognizable**) if it is a recursively enumerable subset in the set of all possible words over the alphabet of the language, i.e., if there exists a Turing machine which will enumerate all valid strings of the language.

Recursively enumerable languages are known as **type-0** languages in the Chomsky hierarchy of formal languages. All regular, context-free, context-sensitive and recursivelanguages are recursively enumerable.

The class of all recursively enumerable languages is called **RE**.

There are three equivalent definitions of a recursively enumerable language:

A recursively enumerable language is a recursively enumerable subset in the set of all possible words over the alphabet of the language.

A recursively enumerable language is a formal language for which there exists a Turing machine (or other computable function) which will enumerate all valid

strings of the language. Note that if the language is infinite, the enumerating algorithm provided can be chosen so that it avoids repetitions, since we can test whether the string produced for number *n* is "already" produced for a number which is less than *n*. If it already is produced, use the output for input *n+1* instead (recursively), but again, test whether it is "new".

A recursively enumerable language is a formal language for which there exists a Turing machine (or other computable function) that will halt and accept when presented with any string in the language as input but may either halt and reject or loop forever when presented with a string not in the language. Contrast this to recursive languages, which require that the Turing machine halts in all cases.

All regular, context-free, context-sensitive and recursive languages are recursively enumerable.

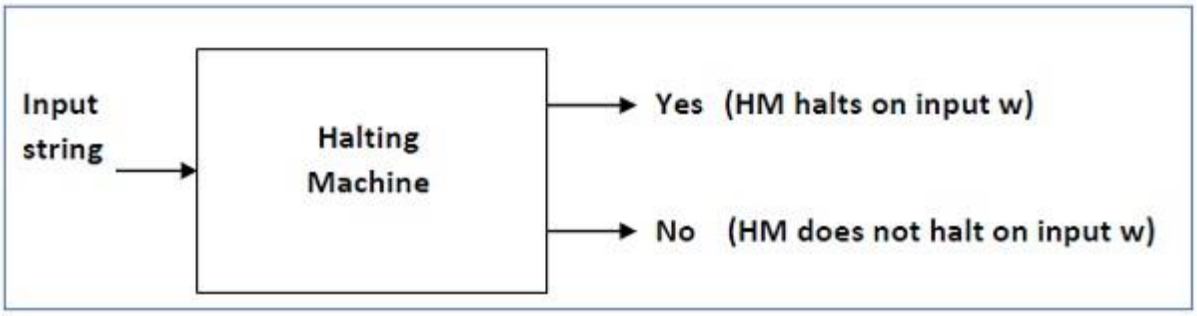
Post's theorem shows that **RE**, together with its complement co-RE, correspond to the first level of the arithmetical hie

3. Halting problems

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.

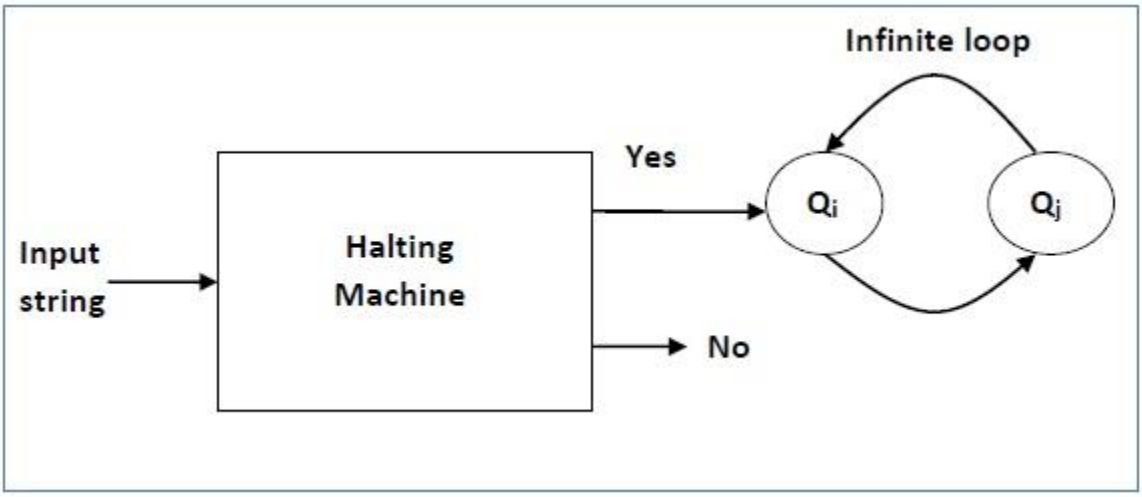
Proof – At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a **Halting machine** that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount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

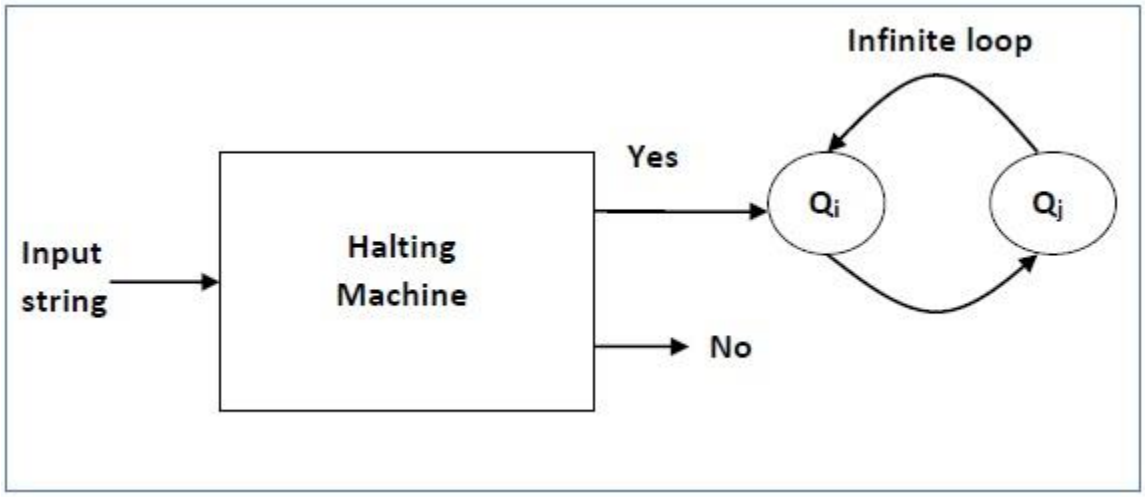
The following is the block diagram of an 'Inverted halting machine' –

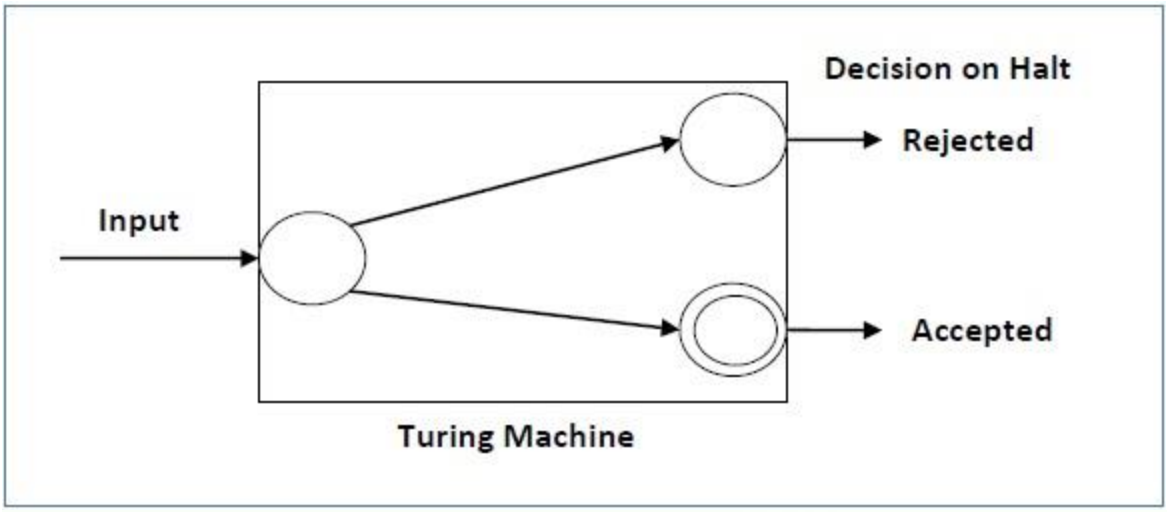


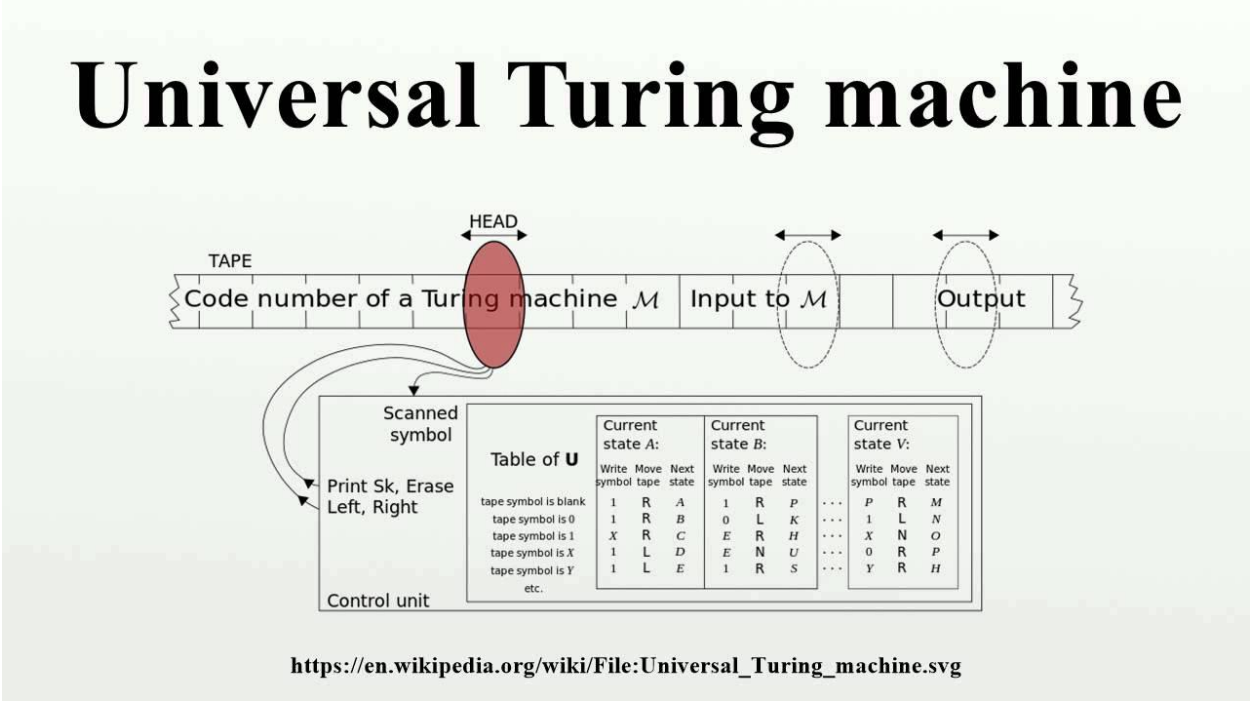
Further, a machine **$(HM)_2$** which input itself is constructed as follows –

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

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







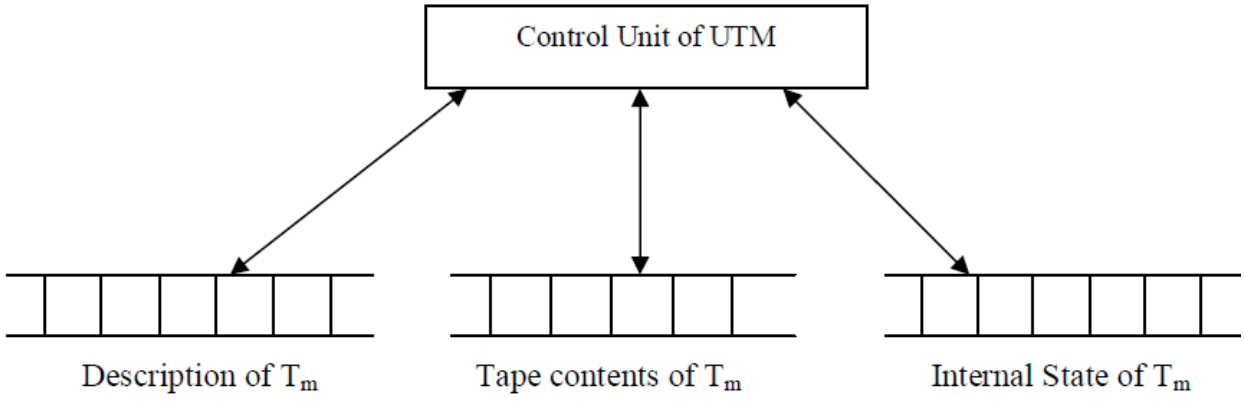
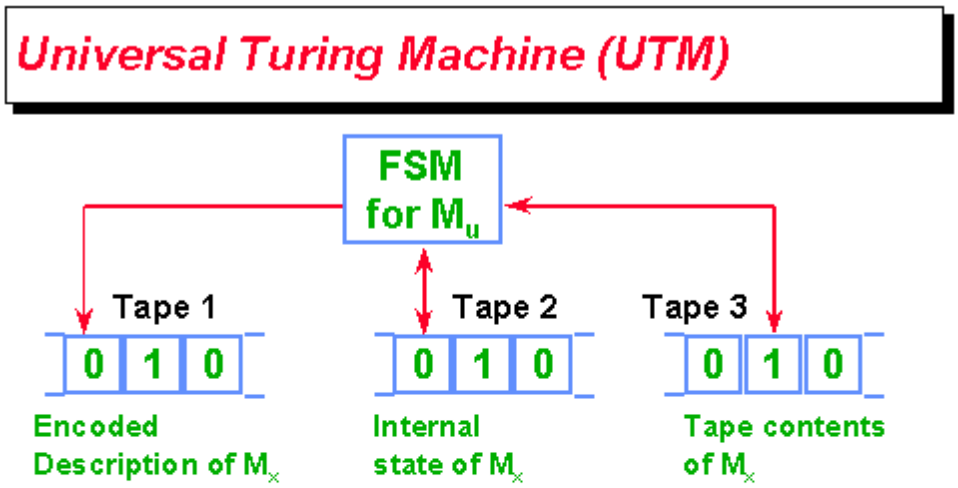


Fig. 1: A Universal Turing Machine



4. Post correspondence problem

The Post Correspondence Problem (PCP), introduced by Emil Post in 1946, is an undecidable decision problem. The PCP problem over an alphabet ∑ is stated as follows –

Given the following two lists, **M** and **N** of non-empty strings over ∑ –

M = (x_1, x_2, x_3,……, x_n)

N = (y_1, y_2, y_3,……, y_n)

We can say that there is a Post Correspondence Solution, if for some i_1,i_2,…… i_k, where $1 \le i_j \le n$, the condition x_{i1} …….x_{ik} = y_{i1} …….y_{ik} satisfies.

Example 1

Find whether the lists

M = (abb, aa, aaa) and N = (bba, aaa, aa)

have a Post Correspondence Solution?

Solution

	x_1	x_2	x_3
M	Abb	aa	aaa
N	Bba	aaa	aa

Here,

$x_2x_1x_3$ = 'aaabbaaa'

and **$y_2y_1y_3$ = 'aaabbaaa'**

We can see that

$x_2x_1x_3 = y_2y_1y_3$

Hence, the solution is **i = 2, j = 1, and k = 3.**

Example 2

Find whether the lists **M = (ab, bab, bbaaa)** and **N = (a, ba, bab)** have a Post Correspondence Solution?

Solution

	x_1	x_2	x_3
M	ab	bab	bbaaa
N	a	ba	bab

In this case, there is no solution because –

$| x_2x_1x_3 | \neq | y_2y_1y_3 |$ (Lengths are not same)

Hence, it can be said that this Post Correspondence Problem is **undecidable**.

Post Correspondence Problem

- FORMAL DEFINITION
  - Given two lists of strings A and B (equal length)

$A = w_1, w_2, \ldots, w_k \qquad B = x_1, x_2, \ldots, x_k$

The problem is to determine if there is a sequence of one or more integers $i_1, i_2, \ldots, i_m$ such that:

$$w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}$$

(w_i, x_i) is called a corresponding pair.

DFA vs NDFA

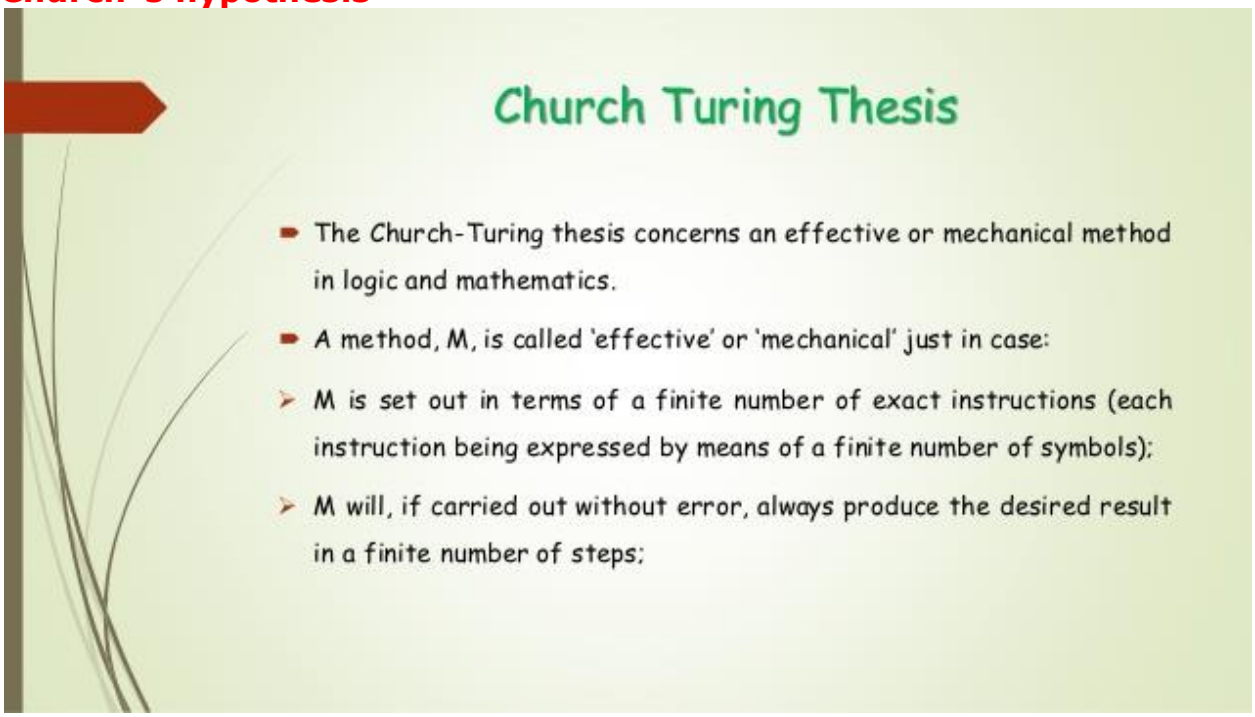
The following table lists the differences between DFA and NDFA.

DFA	NDFA
The transition from a state is to a single particular next state for each input symbol. Hence it is called *deterministic*.	The transition from a state can be to multiple next states for each input symbol. Hence it is called *non-deterministic*.
Empty string transitions are not seen in DFA.	NDFA permits empty string transitions.
Backtracking is allowed in DFA	In NDFA, backtracking is not always possible.
Requires more space.	Requires less space.
A string is accepted by a DFA, if it transits to a final state.	A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state.

5. Ackerman function

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

6. Church"s hypothesis



In computability theory, the **Church–Turing thesis** (also known as **computability thesis**,[1] the **Turing–Church thesis**,[2] the **Church–Turing conjecture**, **Church's thesis**, **Church's conjecture**, and **Turing's thesis**) is a hypothesis about the nature of computable functions. It states that a function on the natural numbers is computable by a human being following an algorithm, ignoring resource limitations, if and only if it is computable by a Turing machine. The thesis is named after American mathematician Alonzo Church and the British mathematician Alan Turing. Before the precise definition of computable function, mathematicians often used the informal term effectively calculable to describe functions that are computable by paper-and-pencil methods. In the 1930s, several independent attempts were made to formalize the notion of computability:

- In 1933, Austrian-American mathematician Kurt Gödel, with Jacques Herbrand, created a formal definition of a class called general recursive functions. The class of general recursive functions is the smallest class of functions (possibly with more than one argument) which includes all constant functions, projections, the successor function, and which is closed under function composition, recursion, and minimization.
- In 1936, Alonzo Church created a method for defining functions called the λ-calculus. Within λ-calculus, he defined an encoding of the natural

numbers called the Church numerals. A function on the natural numbers is called λ-computable if the corresponding function on the Church numerals can be represented by a term of the λ-calculus.
- Also in 1936, before learning of Church's work, Alan Turing created a theoretical model for machines, now called Turing machines, that could carry out calculations from inputs by manipulating symbols on a tape. Given a suitable encoding of the natural numbers as sequences of symbols, a function on the natural numbers is called Turing computable if some Turing machine computes the corresponding function on encoded natural numbers.

Church[3] and Turing[4] proved that these three formally defined classes of computable functions coincide: a function is λ-computable if and only if it is Turing computable if and only if it is *general recursive*. This has led mathematicians and computer scientists to believe that the concept of computability is accurately characterized by these three equivalent processes. Other formal attempts to characterize computability have subsequently strengthened this belief (see below).

On the other hand, the Church–Turing thesis states that the above three formally-defined classes of computable functions coincide with the *informal* notion of an effectively calculable function. Since, as an informal notion, the concept of effective calculability does not have a formal definition, the thesis, although it has near-universal acceptance, cannot be formally proven

1.2 Formulations of Turing's thesis in terms of numbers

In his 1936 paper, titled "On Computable Numbers, with an Application to the Entscheidungsproblem", Turing wrote:

Although the subject of this paper is ostensibly the computable numbers, it is almost equally easy to define and investigate computable functions … I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. (1936: 58)

Computable numbers are (real) numbers whose decimal representation can be generated progressively, digit by digit, by a Turing machine. Examples are:

- any number whose decimal representation consists of a finite number of digits (e.g., 109, 1.142)
- all rational numbers, such as one-third, two-sevenths, etc.
- some irrational real numbers, such as π and e.

Some real numbers, though, are *un*computable, as Turing proved. Turing's proof pointed to specific examples of uncomputable real numbers, but it is easy to see in a general way that there *must* be real numbers that cannot be computed by any Turing machine, since there are *more* real numbers than there are Turing-machine programs. There can be no more Turing-machine programs than there are whole numbers, since the programs can be counted: 1st program, 2nd

program, and so on; but, as Georg Cantor proved in 1874,there are vastly more real numbers than whole numbers (Cantor 1874).

As Turing said, "it is almost equally easy to define and investigate computable functions": there is, in a certain sense, little difference between a computable number and a computable function. For example, the computable number .14159… (formed of the digits following the decimal point in π, 3.1419…) corresponds to the computable function: $f(1)=1$

, $f(2)=4$, $f(3)=1$, $f(4)=5$, $f(6)=9$

, … .

As well as formulations of Turing's thesis like the one given above, Turing also formulated his thesis in terms of numbers:

[T]he "computable numbers" include all numbers which would naturally be regarded as computable. (Turing 1936: 58)

It is my contention that these operations [the operations of an L.C.M.] include all those which are used in the computation of a number. (Turing 1936: 60)

In the first of these two formulations, Turing is stating that every number which is able to be calculated by an effective method (that is, "all numbers which would naturally be regarded as computable") is included among the numbers whose decimal representations can be written out progressively by one or another Turing machine. In the second, Turing is saying that the operations of a Turing machine include all those that a human mathematician needs to use when calculating a number by means of an effective method.

1.3 The meaning of 'computable' and 'computation' in Turing's thesis

Turing introduced his machines with the intention of providing an idealized description of a certain human activity, the tedious one of *numerical computation*. Until the advent of automatic computing machines, this was the occupation of many thousands of people in business, government, and research establishments. These human rote-workers were in fact called *computers*. Human computers used effective methods to carry out some aspects of the work nowadays done by electronic computers. The Church-Turing thesis is about computation *as this term was used in 1936*, viz. human computation (to read more on this, turn to the section 3).

For instance, when Turing says that the operations of an L.C.M. include all those needed "in the computation of a number", he means "in the computation of a number by a human being", since that is what computation was in those days. Similarly "numbers which would naturally be regarded as computable" are numbers which would be regarded as computable by a human computer, a human being who is working solely in accordance with an effective method

UNIT VI:

1. Recursive Function: Basic functions and operations on them

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have –

- **Base criteria** – There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** – The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

- Recursive function theory, like the theory of Turing machines, is one way to make formal and precise the intuitive, informal, and imprecise notion of an effective method. It happens to identify the very same class of functions as those that are Turing computable. This fact is informal or inductive support for Church's Thesis, asserting that every function that is effectively computable in the intuitive sense is computable in these formal ways.
- Like the theory of Turing machines, recursive function theory is vitally concerned with the converse of Church's Thesis: to prove that all the functions they identify as computable are effectively computable, that is, to make clear that they use only effective methods.
- On most lists of "ingredient" functions are the zero function, the successor function, and the identity function.
- z(x) = 0
  s(x) = successor of x (roughly, "x + 1")
  id(x) = x
- The zero function returns zero regardless of its argument. It is hard to imagine a more intuitively computable function.
- The successor function returns the successor of its argument. Since successorship is a more primitive notion than addition, the "x + 1" that I've used to elucidate the function is simply for the convenience of readers. The addition function does not yet exist and must be built up later.
- If successorship is obscure or non-intuitive in the absence of addition, remember that we have allowed ourselves to use the system of natural numbers. We can't calculate yet, but we can count, and that is all we need here. (Hence, the core of recursive function theory that must be intuitive includes not only simple functions and their building operations, but also the natural numbers.)
- The zero and successor functions take only one argument each. But the identity function is designed to take any number of arguments. When it takes one argument (as above) it returns its argument as its value. Again, its effectiveness is obvious. When it takes more than one argument, it returns one of them.

- $id_{2,1}(x,y) = x$
  $id_{2,2}(x,y) = y$
- The two subscripts to the identity function indicate, first, the number of arguments it takes, and second, which argument will be returned by the function. (We could write the identity function more generally so that it applied to any number of arguments, but to preserve simplicity we will not.)

- **The Building Operations**

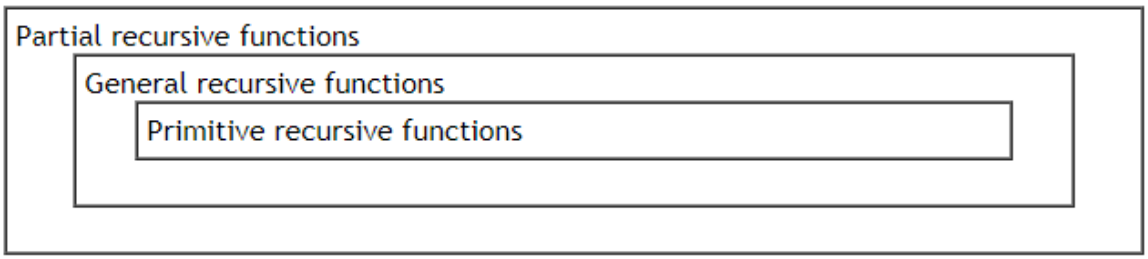
- We will build more complex and interesting functions from the initial set by using only three methods. If we use the analogy to axiomatics again, the methods for building higher functions from our initial set correspond to rules of inference applied to axioms.
- The three methods are composition, primitive recursion, and minimization.
- **Composition**
- If we start with the successor function,
- s(x)

- then we may replace its argument, x, with a function. If we replace the argument, x, with the zero function,
- z(x)

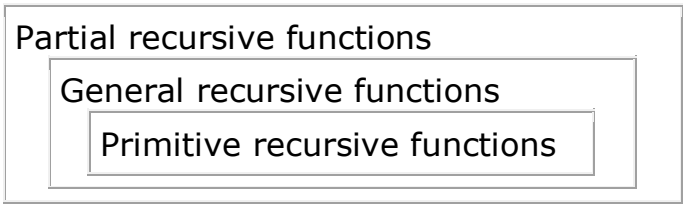
- then the result is the successor of zero.
- s(z(x)) = 1
  s(s(z(x))) = 2 and so on.

- In this way, *compounding* some of the initial functions can describe the natural numbers.
- This building operation is called "composition" (sometimes "substitution"). It rests on the simple fact that computable functions have values, which are numbers. Hence, where we would write a number (numeral) we can instead write any function with its argument(s) that computes that number. In particular, we may use a function with its argument(s) in the place of an argument.
- If we think of a function and its arguments as an alternate *name* of its value, then composition simply allows us to use this kind of name as the argument to a second function.
- In the language used in predicate logic, a function with its arguments is a *term*, and may be used wherever terms are used in predicate logic wffs. The arguments of functions are also terms. Hence, a function can have other functions as arguments.
- We can define the function that adds 2 to a given number thus: add2(x) = s(s(x)). Introducing new names to abbreviate the compound functions created by composition is a great convenience since complex functions are made more intuitive by chunking their simpler components. But the essence of composition is not the re-naming or abbreviating of compound functions, but the compounding itself.
- **Primitive Recursion**

- The second building operation is called primitive recursion. To those not used to it, it can be far from intuitive; to those who have practiced with it, it is fundamental and elegant.
- Function h is defined through functions f and g by primitive recursion when
- h(x,0) = f(x)
  h(x,s(y)) = g(x,h(x,y))
- Let's unpack this slowly. First, remember that f and g are known computable functions. Primitive recursion is a method of defining a new function, h, through old functions, f and g. Second, notice that there are two equations. When h's second argument is zero, the first equation applies; when it is not zero, we use the second. Notice how the successor function enforces the condition that the argument be greater than zero in the second equation. Hence, the first equation applies in the "minimal case" and the second applies in every other case. (It is not unusual for a function to be defined by a series of equations, rather than just one, in order to show its value for relevantly different inputs or arguments.)
- So when (1) the second argument of h is zero, the function is equivalent to some known function f and we compute it; otherwise (2) it is equivalent to some known function g and we compute it. That's how we know that function h, or more generally, the functions built by primitive recursion, will be computable.



- Some terms will help us ring a sub-total. If we allow ourselves to use composition, primitive recursion, and bounded minimization as our building operations, then the functions we can build are called *primitive recursive* (even if we did not use the building operation of primitive recursion). If we add unbounded minimization to our toolkit, we can build a larger class of functions that are called *general recursive* or just plain *recursive*. More precisely, functions are general recursive only when they use unbounded minimization that happens to terminate. Functions using unbounded minimization that does not terminate are called *partial recursive* because they are partial functions.
- Church's Thesis refers to general recursive functions, not to primitive recursive functions. There are effectively computable functions that are not primitive recursive, but that are general recursive (e.g. Ackermann's function). But the open challenge latent in Church's Thesis has elicited no case of an effectively computable function that is not general recursive (if general recursive subsumes and includes the primitive recursive).



- The set of general recursive functions is demonstrably the same as the set of Turing computable functions. Church's Thesis asserts indemonstrably that it is the same as the set of intuitively computable functions. In Douglas Hofstadter's terms, the partial recursive functions are the BlooP computable functions, the general recursive functions are the terminating FlooP computable functions. In Pascal terms, the primitive recursive functions are the FOR loop computable functions, while the general recursive functions the terminating WHILE and REPEAT loop computable functions. The partial recursive functions that are not also general or primitive recursive are the WHILE and REPEAT (FlooP) functions that loop forever without terminating.

Minimization

Let us take a function f(x). Suppose there is at least one value of x which makes f(x) = 0. Now suppose that we wanted to find the *least* value of x which made f(x) = 0. There is an obviously effective method for doing so. We know that x is a natural number. So we set x = 0, and then compute f(x); if we get 0 we stop, having found the least x which makes f(x) = 0; but if not, we try the next natural number 1. We try 0,1,2,3... until we reach the first value which makes f(x) = 0. When we know that *there is* such a value, then we know that this method will terminate in a finite time with the correct answer. If we say that g(x) is a function that computes the least x such that f(x) = 0, then we know that g is computable. We will say that g is produced from f by minimization.

The only catch is that we don't always know that there is any x which makes f(x) = 0. Hence the project of testing 0,1,2,3... may never terminate. If we run the test anyway, that is called *unbounded minimization*. Obviously, when g is produced by unbounded minimization, it is not effectively computable.

If we don't know whether there is any x which makes f(x) = 0, then there is still a way to make function g effectively computable. We can start testing 0,1,2,3... but set an upper bound to our search. We search until we hit the upper bound and then stop. If we find a value before stopping which makes f(x) = 0, then g returns that value; if not, then g returns 0. This is called *bounded minimization*.

(We could define functions f and g more generally so that each took any number of arguments, but again for simplicity we will not.)

Of course, as a building operation we must build only with computable functions. So we build g by minimization only if f(x) is already known to be computable.

While unbounded minimization has the disadvantages of a partial function which may never terminate, bounded minimization has the disadvantage of sometimes failing to minimize.

If you know any programming language, we can make a helpful comparison. Recall the distinction between loops that iterate a definite number of times (e.g. Pascal FOR loops) and those that iterate indefinitely until some special condition is met (e.g. Pascal REPEAT and WHILE loops). Let loops of these two kinds be given the task of testing natural numbers in search of the least value of x in the function f(x) = 0. For each given candidate number, n, starting with 0 and moving upwards, the loop checks to see whether f(n) = 0. The loops which iterate a definite number of times will test only a definite, finite number of candidates. To run such a loop is to take so many steps toward the computation of function g(x). This is bounded minimization at work. Indefinitely iterating loops represent unbounded minimization: they will keep checking numbers until they find one which makes the value of function f equal zero, or until the computer dies of old age or we turn it off.

In general we will have no way of knowing in advance whether either kind of loop will ever discover what it is looking for. But we will always know in advance that a bounded search will come to an end anyway, while an unbounded search may or may not. That is the decisive difference for effective computability.

Suppose an unbounded search does terminate with a good value for y. Since the program did halt, it must have halted after a definite, finite number of steps. That means that we could have done the same work with a bounded search, if only we had known how large to make the upper bound. (In fact, we could have done the whole search without loops at all, if only we had known how many times to repeat the incrementing statements.) This fact is more important for theory than for practice. For theory it means that whatever unbounded minimization can actually compute can be computed by bounded minimization; and since the latter is effective, the ineffectiveness of the former is significantly mitigated. (We can increase the bound in bounded minimization to any finite number, in effect covering the ground effectively that unbounded minimization covers ineffectively.) For practical computation this means little, however. Since we rarely know where to set the upper bound, we will use unbounded searches even though that risks non-termination (infinite loops). Our job is not made any easier by knowing that if our unbounded search should ever terminate, then a bounded search could have done the same work without the risk of non-termination.

2. Bounded Minimalization

One useful way of generating more primitive recursive functions from existing ones is through what is known as *bounded summation* and *bounded product*. Given a primitive recursive function $f:\mathbb{N}^{m+1}\to\mathbb{N}$, define two functions $fs,fp:\mathbb{N}^{m+1}\to\mathbb{N}$ as follows: for $\boldsymbol{x}\in\mathbb{N}^m$ and $y\in\mathbb{N}$:

$$fs(\boldsymbol{x},y):=\sum_{i=0}^{y} f(\boldsymbol{x},i)$$

$$fp(\boldsymbol{x},y):=\prod_{i=0}^{y} f(\boldsymbol{x},i)$$

These are easily seen to be primitive recursive, because they are defined by primitive recursion. For example,

$$fs(\boldsymbol{x},0)=f(\boldsymbol{x},0)$$
$$fs(\boldsymbol{x},n+1)=g(\boldsymbol{x},n,fs$$

where $g(\boldsymbol{x},n,y)=add(f(\boldsymbol{x},n),y)$, which is primitive recursive by functional composition.

Definition. We call fs and fp functions obtained from f by *bounded sum* and *bounded product* respectively.

Using bounded summation and bounded product, another useful class of primitive recursive functions can be generated:

Definition. Let $f:\mathbb{N}^{m+1}\to\mathbb{N}$ be a function. For each $y\in\mathbb{N}$, set

$$Af(\boldsymbol{x},y):=\{z\in\mathbb{N}\mid z\le y \text{ and }$$

Define

$$fbmin(\boldsymbol{x},y):=\{\min Af(\boldsymbol{x},y) \text{ if } Af$$

$fbmin$ is called the function obtained from f by *bounded minimization*, and is usually denoted

$$\mu z\le y(f(\boldsymbol{x},z)=0$$

Proposition 1.

If $f:N^{m+1}\to N$ is primitive recursive, so is $fbmin$.

Proof.

Define $g:=sgn\circ f$. Then

g⁢(𝒙,y):={0if ⁢f⁢(𝒙,y)=0,1otherwise.

As f is primitive recursive, so is g, since the sign function sgn is primitive recursive (see this entry (http://planetmath.org/ExamplesOfPrimitiveRecursiveFunctions)).

Next, the function gp obtained from g by bounded product has the following properties:

if gp⁢(𝒙,y)=1, then gp⁢(𝒙,z)=1 for all z<y,

if gp⁢(𝒙,y)=0, then gp⁢(𝒙,z)=0 for all z≥y.

Finally, the function (gp)s obtained from gp by bounded sum has the property that, when applied to (𝒙,y), counts the number of z≤y such that gp⁢(𝒙,z)=1. Based on the property of gp, this count is then exactly the least z≤y such that gp⁢(𝒙,z)=1. This means that (gp)s=fb⁢m⁢i⁢n for all (𝒙,y)∈ℕm+1. Since gp is primitive recursive, so is (gp)s, or that fb⁢m⁢i⁢n is primitive recursive. ■

In fact, if f is a (total) recursive function, so is fb⁢m⁢i⁢n, because all of the derived functions in the proof above preserve primitive recursiveness as well as totalness.

Remarks.

In the definition of bounded minimization, if we take the y out, then we arrive at the notion of *unbounded minimization*, or just *minimization*.
The proposition above shows that the set$\mathcal{P}\mathcal{R}$ of primitive recursive functions is closed under bounded minimization. However, $\mathcal{P}\mathcal{R}$ is not closed under minimization. The closure of $\mathcal{P}\mathcal{R}$ under minimization is the set $\mathcal{R}$ of recursive functions (total or not).

It is not hard to show that $\mathcal{E}\mathcal{R}$, the set of all elementary recursive functions, is closed under bounded minimization.

3. Unbounded Minimalization,

4. Primitive recursive function

In computability theory, **primitive recursive functions** are a class of functions that are defined using composition and **primitive recursion** – described below – as central operations. They are a strict subset of those μ-recursive functions (also called partial recursive functions) which are also total

functions. Primitive recursive functions form an important building block on the way to a full formalization of computability. These functions are also important in proof theory.

Most of the functions normally studied in number theory are primitive recursive. For example, addition and division, the factorial and exponential function, and the function which returns the nth prime are all primitive recursive. So are many approximations to real-valued functions.[1] In fact, it is difficult to devise a total recursive function that is *not* primitive recursive, although some are known (see the section on Limitations below). The set of primitive recursive functions is known as PR in computational complexity theory

The primitive recursive functions are among the number-theoretic functions, which are functions from the natural numbers (nonnegative integers) {0, 1, 2, ...} to the natural numbers. These functions take n arguments for some natural number n and are called n-ary.

The basic primitive recursive functions are given by these axioms:

1. **Constant function**: The 0-ary constant function 0 is primitive recursive.
2. **Successor function**: The 1-ary successor function S, which returns the successor of its argument (see Peano postulates), is primitive recursive. That is, $S(k) = k + 1$.
3. **Projection function**: For every $n \geq 1$ and each i with $1 \leq i \leq n$, the n-ary projection function P_i^n, which returns its i-th argument, is primitive recursive.

More complex primitive recursive functions can be obtained by applying the operations given by these axioms:

4. **Composition**: Given a k-ary primitive recursive function f, and k many m-ary primitive recursive functions $g_1,...,g_k$, the composition of f with $g_1,...,g_k$,

   i.e. the m-ary function is primitive recursive.

Example. We take $f(x)$ as the $S(x)$ defined above. This f is a 1-ary primitive recursive function. And so is $g(x) = S(x)$. So $h(x)$ defined as $f(g(x)) = S(S(x))$ is a primitive recursive 1-ary function too. Informally speaking, $h(x)$ is the function that turns x into $x+2$.

5. **Primitive recursion**: Given f, a k-ary primitive recursive function, and g, a $(k+2)$-ary primitive recursive function, the $(k+1)$-ary function h is defined as the primitive recursion of f and g, i.e. the function h is primitive recursive when

   and

Example. Suppose $f(x) = P_1^1(x) = x$ and $g(x,y,z) = S(P_2^3(x,y,z)) = S(y)$. Then $h(0,x) = x$ and $h(S(y),x) = g(y,h(y,x),x) = S(h(y,x))$. Now $h(0,1) = 1$, $h(1,1) = S(h(0,1)) = 2$, $h(2,1) = S(h(1,1)) = 3$. This h is a 2-ary primitive recursive function. We can call it 'addition'.

The **primitive recursive** functions are the basic functions and those obtained from the basic functions by applying these operations a finite number of times.

Role of the projection functions

The projection functions can be used to avoid the apparent rigidity in terms of the arity of the functions above; by using compositions with various projection functions, it is possible to pass a subset of the arguments of one function to another function. For example, if g and h are 2-ary primitive recursive functions then

is also primitive recursive. One formal definition using projection functions is

Converting predicates to numeric functions

In some settings it is natural to consider primitive recursive functions that take as inputs tuples that mix numbers with truth values (that is t for true and f for false), or that produce truth values as outputs.[2] This can be accomplished by identifying the truth values with numbers in any fixed manner. For example, it is common to identify the truth value t with the number 1 and the truth value f with the number 0. Once this identification has been made, the characteristic function of a set A, which always returns 1 or 0, can be viewed as a predicate that tells whether a number is in the set A. Such an identification of predicates with numeric functions will be assumed for the remainder of this article.

Computer language definition

An example of a primitive recursive programming language is one that contains basic arithmetic operators (e.g. + and −, or ADD and SUBTRACT), conditionals and comparison (IF-THEN, EQUALS, LESS-THAN), and bounded loops, such as the basic for loop, where there is a known or calculable upper bound to all loops (FOR i FROM 1 to n, with neither i nor n modifiable by the loop body). No control structures of greater generality, such as while loops or IF-THEN plus GOTO, are admitted in a primitive recursive language. Douglas Hofstadter's BlooP in *Gödel, Escher, Bach* is one such. Adding unbounded loops (WHILE, GOTO) makes the

language partially recursive, or Turing-complete; Floop is such, as are almost all real-world computer languages.

Arbitrary computer programs, or Turing machines, cannot in general be analyzed to see if they halt or not (the halting problem). However, all primitive recursive functions halt. This is not a contradiction; primitive recursive programs are a non-arbitrary subset of all possible programs, constructed specifically to be analyzable.

Examples

Most number-theoretic functions definable using recursion on a single variable are primitive recursive. Basic examples include the addition and truncated subtraction functions.

Addition

Intuitively, addition can be recursively defined with the rules:

,

To fit this into a strict primitive recursive definition, define:

Here S(n) is "the successor of n" (i.e., n+1), P_1^1 is the identity function, and P_2^3 is the projection function that takes 3 arguments and returns the second one. Functions f and g required by the above definition of the primitive recursion operation are respectively played by P_1^1 and the composition of S and P_2^3.

Subtraction

See also: Monus

Because primitive recursive functions use natural numbers rather than integers, and the natural numbers are not closed under subtraction, a truncated subtraction function (also called "proper subtraction") is studied in this context. This limited subtraction function sub(a, b) [or $b \dot{-} a$] returns b - a if this is nonnegative and returns 0 otherwise.

The **predecessor function** acts as the opposite of the successor function and is recursively defined by the rules:

pred(0) = 0,

$\text{pred}(n + 1) = n$.

These rules can be converted into a more formal definition by primitive recursion:

$\text{pred}(0) = 0$,
$\text{pred}(S(n)) = P_1^2(n, \text{pred}(n))$.

The limited subtraction function is definable from the predecessor function in a manner analogous to the way addition is defined from successor:

$\text{sub}(0, x) = P_1^1(x)$,
$\text{sub}(S(n), x) = \text{pred}(P_2^3(n, \text{sub}(n, x), x))$.

Here sub(a, b) corresponds to $b \dot{-} a$; for the sake of simplicity, the order of the arguments has been switched from the "standard" definition to fit the requirements of primitive recursion. This could easily be rectified using composition with suitable projections.

Other operations on natural numbers

Exponentiation and primality testing are primitive recursive. Given primitive recursive functions e, f, g, and h, a function that returns the value of g when $e \leq f$ and the value of h otherwise is primitive recursive.

Operations on integers and rational numbers

By using Gödel numberings, the primitive recursive functions can be extended to operate on other objects such as integers and rational numbers. If integers are encoded by Gödel numbers in a standard way, the arithmetic operations including addition, subtraction, and multiplication are all primitive recursive. Similarly, if the rationals are represented by Gödel numbers then the field operations are all primitive recursive.

Use in first-order Peano arithmetic

In first-order Peano arithmetic, there are infinitely many variables (0-ary symbols) but no k-ary non-logical symbols with k>0 other than S, +, *, and ≤. Thus in order to define primitive recursive functions one has to use the following trick by Gödel.

By using a Gödel numbering for sequences, for example Gödel's β function, any sequence of numbers can be encoded by a single number. Such a number can therefore represent the primitive recursive function until a given n.

Let h be a 1-ary primitive recursion function defined by:

where C is a constant and g is an already defined function.

Using Gödel's β function, for any sequence of natural numbers (k_0, k_1, …, k_n), there are natural numbers b and c such that, for every i ≤ n, β(b, c, i) = k_i. We may thus use the following formula to define *h*; more precisely, *m*=*h*(*n*) is a

shorthand for the following:

and the equating to *g*, being already defined, is in fact shorthand for some other already defined formula (as is β, whose formula is given here).

The generalization to any k-ary primitive recursion function is trivial.

Relationship to recursive functions

The broader class of partial recursive functions is defined by introducing an unbounded search operator. The use of this operator may result in a partial function, that is, a relation with *at most* one value for each argument, but does not necessarily have *any* value for any argument (see domain). An equivalent definition states that a partial recursive function is one that can be computed by a Turing machine. A total recursive function is a partial recursive function that is defined for every input.

Every primitive recursive function is total recursive, but not all total recursive functions are primitive recursive. The Ackermann function *A*(*m*,*n*) is a well-known example of a total recursive function (in fact, provable total), that is not primitive recursive. There is a characterization of the primitive recursive functions as a subset of the total recursive functions using the Ackermann function. This characterization states that a function is primitive recursive if and only if there is a natural number *m* such that the function can be computed by a Turing machine that always halts within A(*m*,*n*) or fewer steps, where *n* is the sum of the arguments of the primitive recursive function.[3]

An important property of the primitive recursive functions is that they are a recursively enumerable subset of the set of all total recursive functions (which is not itself recursively enumerable). This means that there is a single computable function *f*(*m*,*n*) that enumerates the primitive recursive functions, namely:

- For every primitive recursive function *g*, there is an *m* such that *g*(*n*) = *f*(*m*,*n*) for all *n*, and
- For every *m*, the function *h*(*n*) = *f*(*m*,*n*) is primitive recursive.

f can be explicitly constructed by iteratively repeating all possible ways of creating primitive recursive functions. Thus, it is provably total. One can use a diagonalization argument to show that *f* is not recursive primitive in itself: had it been such, so would be *h*(*n*) = *f*(*n*,*n*)+1. But if this equals some primitive recursive function, there is an *m* such that *h*(*n*) = *f*(*m*,*n*) for all *n*, and then *h*(*m*) = *f*(*m*,*m*), leading to contradiction.

However, the set of primitive recursive functions is not the *largest* recursively enumerable subset of the set of all total recursive functions. For example, the set of provably total functions (in Peano arithmetic) is also recursively enumerable, as one can enumerate all the proofs of the theory. While all primitive recursive functions are provably total, the converse is not true.

5. μ-recursive function

In mathematical logic and computer science, the **μ-recursive functions** are a class of partial functions from natural numbers to natural numbers which are "computable" in an intuitive sense. In fact, in computability theory it is shown that the μ-recursive functions are precisely the functions that can be computed by Turing machines. The μ-recursive functions are closely related to primitive recursive functions, and their inductive definition (below) builds upon that of the primitive recursive functions. However, not every μ-recursive function is a primitive recursive function — the most famous example is the Ackermann function.

Other equivalent classes of functions are the λ-recursive functions and the functions that can be compute

The **μ-recursive functions** (or **partial μ-recursive functions**) are partial functions that take finite tuples of natural numbers and return a single natural number. They are the smallest class of partial functions that includes the initial functions and is closed under composition, primitive recursion, and the μ operator.

The smallest class of functions including the initial functions and closed under composition and primitive recursion (i.e. without minimisation) is the class of primitive recursive functions. While all primitive recursive functions are total, this is not true of partial recursive functions; for example, the minimisation of the successor function is undefined. The set of total recursive functions is a subset of the partial recursive functions and is a superset of the primitive recursive functions; functions like the Ackermann function can be proven to be total recursive, and not primitive.

The first three functions are called the "initial" or "basic" functions: (In the following the subscripting is per Kleene (1952) p. 219. For more about some of the various symbolisms found in the literature see Symbolism below.)

1. **Constant function**: For each natural number $n$and every k:

   $f(x_1, \ldots, x_k) = n$.

   Alternative definitions use compositions of the successor function and use a **zero function**, that always returns zero, in place of the constant function.

2. **Successor function S:**

$$S(x) \stackrel{\text{def}}{=} f(x) = x + 1$$

3. **Projection function** P_i^k (also called the **Identity function** I_i^k): For all natural numbers i, k such that $1 \leq i \leq k$:

$$P_i^k \stackrel{\text{def}}{=} f(x_1, \ldots, x_k) = x_i.$$

4. **Composition operator** $\circ$ (also called the **substitution operator**): Given an m-ary function $h(x_1, \ldots, x_m)$ and m k-ary functions $g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k)$:

$$h \circ (g_1, \ldots, g_m) \stackrel{\text{def}}{=} f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k)).$$

5. **Primitive recursion operator** ρ: Given the k-ary function $g(x_1, \ldots, x_k)$ and k+2 -ary function $h(y, z, x_1, \ldots, x_k)$:

$$\begin{aligned} \rho(g, h) &\stackrel{\text{def}}{=} f(y, x_1, \ldots, x_k) \quad \text{where} \\ f(0, x_1, \ldots, x_k) &= g(x_1, \ldots, x_k) \\ f(y+1, x_1, \ldots, x_k) &= h(y, f(y, x_1, \ldots, x_k), x_1, \ldots, x_k). \end{aligned}$$

6. **Minimisation operator** μ: Given a (k+1)-ary total function $f(y, x_1, \ldots, x_k)$:

$$\begin{aligned} \mu(f)(x_1, \ldots, x_k) = z \stackrel{\text{def}}{\iff} & \exists y_0, \ldots, y_z \quad \text{such that} \\ & y_i = f(i, x_1, \ldots, x_k) \quad \text{for } i = 0, \ldots, z \\ & y_i > 0 \quad \text{for } i = 0, \ldots, z-1 \\ & y_z = 0 \end{aligned}$$

Intuitively, minimisation seeks--beginning the search from 0 and proceeding upwards--the smallest argument that causes the function to return zero; if there is no such argument, the search never terminates.

6. Primitive recursive predicates

In this entry, we give some examples of primitive recursive predicates. In particular, we will show that the predicate "x is a prime number" is primitive recursive.

In the examples, we use $\Phi(x)$ to indicate a predicate (over the natural numbers $\mathbb{N}$). First, two very simple examples:

1. 1.

   $\Phi(x)$ given by "$x=n$" for a fixed $n\in\mathbb{N}$ is primitive recursive. To see this, note that the set associated with Φ is $\{n\}$, and its associated characteristic function is primitive recursive by example 3(l) in this entry (http://planetmath.org/ExamplesOfPrimitiveRecursiveFunctions). As a result, Φ is primitive recursive.

2. 2.

   $\Phi(x,y)$ given by "$x\leq y$" is primitive recursive. The associated set is $\{(x,y)|x\leq y\}$ whose characteristic function is $sgn(s(y)\dot{-}x)$, which is primitive recursive.

Before exhibiting more examples, we prove some basic facts about primitive recursive predicates:

Proposition 1.

If Φ and Θ are primitive recursive predicates, then so are $\neg\Phi$ (negation), $\Phi\vee\Theta$ (disjunction), and $\Phi\wedge\Theta$ (conjunction).

Proof.

Let S,T be the sets associated with the predicates Φ and Θ. We may assume that $S,T\subseteq\mathbb{N}k$ for some positive integer k. Let cS and cT be the associated characteristic functions.

First, $1\dot{-}cS$ is the characteristic function of $\mathbb{N}k-S$, which is the associated set of the predicate $\neg\Phi$. Since $1\dot{-}cS$ is primitive recursive, so is $\neg\Phi$.

Next, mult(CS,CT) is the characteristic function of $S\cap T$, which is the associated set of the predicate $\Phi\wedge\Theta$. Since mult(CS,CT) is primitive recursive, so is $\Phi\wedge\Theta$.

Finally, the set associated with $\Phi\vee\Theta$ is $S\cup T$, which is $\mathbb{N}k-((\mathbb{N}k-S)\cap(\mathbb{N}k-T))$, which is primitive recursive. Hence $\Phi\vee\Theta$ is primitive recursive as well. ■

In short, primitive recursiveness is preserved under Boolean operations on predicates. By induction, we also see primitive recursiveness is preserved under finite disjunction and finite conjunction.

Using this fact, we list three more examples:

1. 3.

   the predicate "$x=y$" is primitive recursive, since it is the conjunction of primitive recursive predicates "$x\leq y$" and "$y\leq x$".

2. 4.

   the predicate "y<x" is primitive recursive, since it is the conjunction "y≤x" and "¬(x=y)", both of which are primitive recursive.

3. 5.

   the predicate "x∈S", where S is a fixed finite set, is primitive recursive, as it is the disjunction of primitive predicates of the form "x=n", where n ranges over S. Since the disjunction operation is finitary, "x∈S" is primitive recursive also.

Here's another useful fact about primitive recursive predicates:

Proposition 2.

If Φ(x1,…,xm) is primitive recursive, so is Θ(x1,…,xn), defined by simultaneous substitution of the variables xi in Φ by n-ary primitive recursive functions fi:

$$\Theta(x_1,\ldots,x_n) := \Phi(f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n)).$$

Proof.

Let cΦ be the characteristic function of (the set associated with) Φ, then

$$c\Phi(f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n))$$

is cΘ, the characteristic function of Θ. Since cΦ, fi are primitive recursive, cΘ, obtained by functional composition, and hence Θ, are primitive recursive too. ■

Let us list two more examples:

1. the predicate "x·y≠z" is primitive recursive.
2. the predicate "y<x!" is primitive recursive.

Finite disjunctions and finite conjunctions of predicates can be thought of as special cases of bounded quantification (http://planetmath.org/BoundedQuantifier) on predicates. In view of this, we have the following facts:

Proposition 3.

If $\Phi(\boldsymbol{x},y)$ is primitive recursive, so are

$$\Theta(\boldsymbol{x},y) := \forall z \le y\, \Phi(\boldsymbol{x},z) \quad \textit{and} \quad \Psi(\boldsymbol{x},y) := \exists z \le y\, \Phi(\boldsymbol{x},z).$$

Proof.

The set associated with $\Theta(x,y)$ is $\{(x,y)|$ for all $z\leq y,\Phi(x,z)\}$, which is just

$$S:=\{(x,y)|\Phi(x,0)\}\cap\{(x,y)|\Phi(x,1)\}\cap\cdots\cap\{(x,y)|\Phi(x,y)\}$$

For each fixed i, let $\Phi i(x):=\Phi(x,i)$. Then by proposition 2, Φi is primitive recursive. Let ci be the characteristic function of Φi, and d the function such that $d(x,i)$ is $ci(x)$ if $i\in\{0,\ldots,n\}$, and 0 otherwise. So d is primitive recursive (see this entry (http://planetmath.org/ExamplesOfPrimitiveRecursiveFunctions)). Now, define function c by

$$c(x,y):=\prod i=0yd(x,i).$$

So $1=d(x,i)=ci(x)$ iff $\Phi i(x)$ iff $(x,i)\in\{(x,i)|\Phi(x,i)\}$ iff $(x,y)\in\{(x,y)|\Phi(x,i)\}$.

As a result, $1=c(x,y)$ iff $(x,y)\in\{(x,y)|\Phi(x,i)\}$ for all $i\in\{0,\ldots,y\}$ iff $(x,y)\in S$. In other words, c is the characteristic function of S. Since c is obtained from d by bounded product, c is primitive recursive. This shows that Θ is primitive recursive as well.

Next, since the characteristic function of $\Psi(x,y)$ is the same as the characteristic function of $\neg\forall z\leq y(\neg\Phi(x,z))$, we conclude that Ψ is primitive recursive by proposition 1 and what we have just proved. ■

Our final example is the following:

1. 8.

   Φ(x) given by "x is prime" is primitive recursive. One way to characterize Φ is the following: 1<x, and x can not be written as a product of two natural numbers, both greater than 1. In other words,

   $$\Phi(x)\equiv(1<x)\wedge\forall y(\forall z\neg(x=yz\wedge 1<y\wedge 1<z)),$$

   where ≡ means "can be characterized by" (they share the same characteristic function). Observe that since $x=yz$, both y and z are in fact bounded by x. Therefore,

   $$\Phi(x)\equiv(1<x)\wedge\forall y\leq x(\forall z\leq x\neg(x=yz\wedge 1<y\wedge 1<z)).$$

   As each of $x=yz$, $1<y$, and $1<z$ is primitive recursive, so are their conjunction:

$x=y\boxtimes z\wedge 1<y\wedge 1<z$,

the negation of which:

$\neg\boxdot(x=y\boxtimes z\wedge 1<y\wedge 1<z)$,

two successive applications of bounded universal quantifiers:

$\forall y\leq x(\forall z\leq x\neg(x=yz\wedge 1<y\wedge 1<z))$,

and finally its conjunction with the primitive recursive predicate $1<x$, which is just $\Phi\boxtimes(x)$. Therefore, $\Phi\boxtimes(x)$ is primitive recursive.

7. Mod and Div functions

The mod(x,y) function, or the mod operator x % y in C# or JavaScript say, is very simple but you can think about it at least two different ways - corresponding, roughly speaking, to passive and active models of what is going on.

- First the passive:

  The most obvious definition is:

  mod(x,y) or x % y gives the remainder when you divide x by y

  In this case you are thinking of working out what is left over if you do something in `chunks of y'.

  For example, a text file is 20000 lines long and you can fit 66 lines to a page - how much text is left over? The answer is 2 because mod(20000,66)=20000 % 66 =2.

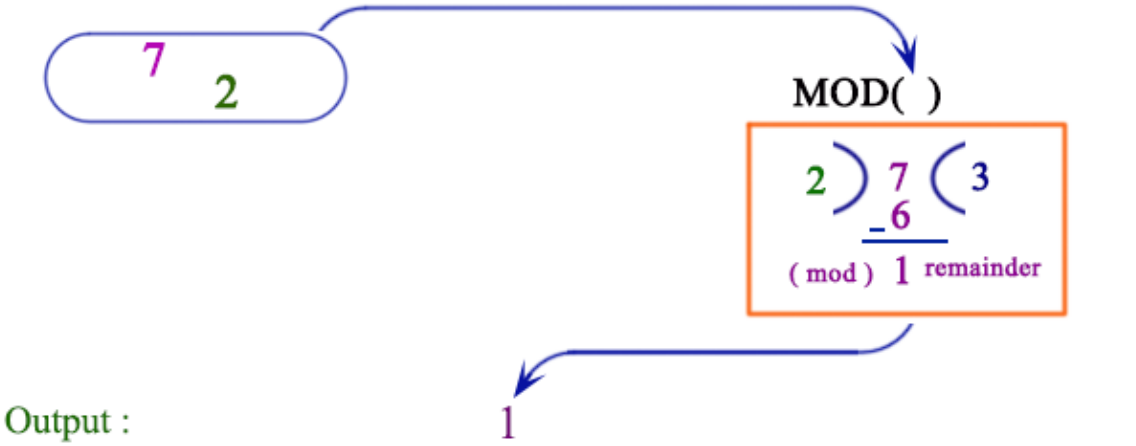
MySQL MOD() function

Syntax :

MOD(N, M)

Example :

MOD(7, 2)



Output :

1

Quotient-Remainder Theory, Div and Mod

If n and d are integers and $d > 0$, then

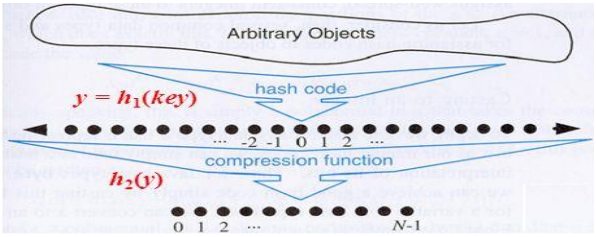
$$n \; div \; d = q \text{ and } n \; mod \; d = r \quad \Leftrightarrow \quad n = dq + r$$

where q and r are integers and $0 \leq r < d$.

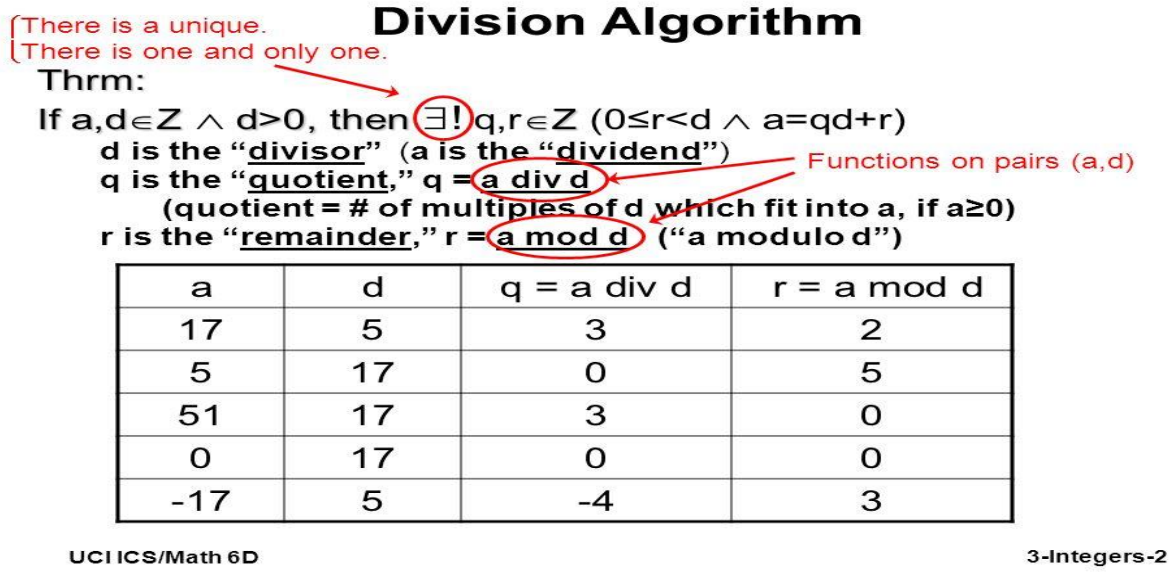
Quotient: $\mathrm{q} = n \; div \; d = \left\lfloor \frac{n}{d} \right\rfloor$

Reminder: $\mathrm{r} = n \; mod \; d = n - dq$

Compression Functions



- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a **prime**
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = [(ay + b) \bmod p] \bmod N$
  - p is a prime number larger than N
  - a and b are integers chosen at random from $[0, p-1]$, with $a > 0$



Qu-
Let $S(i,x1,\ldots,xn)$ be a primitive recursive predicate.
$f(i1,i2,x1,\ldots,xn)$={10 when for all i, $i1\le i\le i2, S(i,x1,\ldots,xn)=1$ otherwise

Show that $f(i1,i2,x1,\ldots,xn)$

is also primitive recursive

Ans- Do not get confused by the word "predicate". It is actually quite common to show that a function is primitive recursive by using another function for which we already know that it is primitive recursive in its definition.

For example, in the proof that multiplication is primitive recursive one usually defines multiplication using addition, which is not a basic primitive recursive function.

If "predicate" confuses you check out its definition, e.g., wikipedia. It is just a function $f:N\to\{0,1\}$

. So you can use a primitive recursive predicate just as a primitive recursive function of that kind.

All you need to show is, as mentioned in the comments, that the bounded quantification is primitive recursive.

Primitive Recursive Function for Division and Hailstone function

Are division and Hailstone primitive recursion function?

Div(x,y)={x/y,0,if y divides x otherwise

Hailstone(n)={$3n+1$,$n/2$,if n is odd if n is even

I tried to solve division in this way

Div(0,y)=0
Div($x+y$,y)=Div(x,y)+1

For your division example, what would the case be if y was equal to zero?

The function is undefined if y = 0. Thus, it is not a total function. Thus, division is not primitively recursive, as all primitively recursive functions are total functions.

> Regarding the first two paragraphs of this answer: The function Div(x,y) *is* defined if y=0 and x>0. We say y divides x iff there exists an integer z such that $yz=x$. Thus, 0 divides x iff x=0. Therefore, when x>0, we have Div(x,0)=0. The only ambiguous case is Div(0,0). I'd guess that the original poster means that Div(0,0)=0. But if there is some doubt in your mind about that, I think it'd be more useful to post a comment requesting clarification, or suggest an edit to the question that defines Div more carefully.
>
> The question was "Are division and Hailstone primitive recursion function(s)?", not "is function X primitively recursive?" The division function is not primitively recursive, period. I don't have to ask him to define for me what division is. A function X I would have to ask - but then that function would not be division

Equivalence of Turing Computable function and μ-recursive function.

In computability theory, computable functions are also called recursive functions. At least at first sight, they do not have anything in common with what you call "recursive" in day-to-day programming (i.e., functions that call themselfes).

What is the actual meaning of recursive in the context of computability? Why are those functions called "recursive"?

Define some basic functions:

- zero function

$$zero:\mathrm{N}\to\mathrm{N}:x\mapsto 0$$

- successor function

$$succ:\mathrm{N}\to\mathrm{N}:x\mapsto x+1$$

- projection function

$$p_{ni}:\mathrm{N}_n \to \mathrm{N}:(x_1,x_2,\ldots,x_n) \mapsto x_i$$

From now on I will use $x_n^{\_}$

to denote $(x_1,x_2,\ldots,x_n)$

Define a composition:

Given functions

- $g_1,g_2,\ldots,g_m$

each with signature $\mathrm{N}_k \to \mathrm{N}$

- $f:\mathrm{N}_m \to \mathrm{N}$
-

Construct the following function:

$$h:\mathrm{N}_k \to \mathrm{N}:x_k^{\_} \mapsto h(x_k^{\_}) = f(g_1(x_k^{\_}),g_2(x_k^{\_}),\ldots,g_m(x_k^{\_}))$$

Define primitive recursion:

Given functions

- $f:\mathrm{N}_k \to \mathrm{N}$
- $g:\mathrm{N}_{k+2} \to \mathrm{N}$

Construct the following (piecewise) function:

$$h:\mathrm{N}_{k+1} \to \mathrm{N}:(x_k^{\_},y+1) \mapsto \{f(x_k^{\_}),g(x_k^{\_},y,h(x_k^{\_},y)),y+1=0y+1>0$$

All functions that can be made using **compositions** and **primitive recursion** on **basic functions**, are called **primitive recursive**. It is called that way by definition. While a link with functions that call themselves exists, there's no need to try and link them with each other. You might consider recursion a homonym.

This definition and construction above was constructed by Gödel (a few other people were involved too) in an attempt to capture all functions that are computable i.e. there exists a Turing Machine for that function. Note that the concept of a Turing Machine was not yet described, or it was at least very vague.

(Un)fortunately, someone called Ackermann came along and defined the following function:

- $Ack:\mathbb{N}^2\rightarrow\mathbb{N}$

- $Ack(0,y)=y+1$
- $Ack(x+1,0)=Ack(x,1)$
- $Ack(x+1,y+1)=Ack(x,Ack(x+1,y))$

This function is computable, but there's no way to construct it using only the constructions above! (i.e. Ack

is not primitive recursive) This means that Gödel and his posse failed to capture all computable functions in their construction!

Gödel had to expand his class of functions so Ack

could be constructed. He did this by defining the following:

Unbounded minimisation

- $g:\mathbb{N}^k\rightarrow\mathbb{N}$

□ □ IF $[f(\bar{x_k},y)=0$ AND $f(\bar{x_k},z)$ is defined $\forall z<y$ AND $f(\bar{x_k},z)\neq 0]$
THEN
$g(\bar{x_k})=y$
ELSE
$g(\bar{x_k})$

- is not defined.

This last one may be hard to grasp, but it basically means that $g((x_1,x_2,...,x_k))$

is the smallest root of f

(if a root exists).

All functions that can be constructed with all the constructions defined above are called **recursive**. Again, the name **recursive** is just by definition, and it doesn't necessarily have correlation with functions that call themselves. Truly, consider it a homonym.

Recursive functions can be either **partial recursive functions** or **total recursive functions**. All partial recursive functions are total recursive functions. All primitive recursive functions are total. As an example of a partial recursive function that is not total, consider the minimisation of the successor function. The successor function doesn't have roots, so its minimisation is not defined. An example of a total recursive function (which uses minimisation) is *Ack*

.

Now Gödel was able to construct the *Ack*

function as well with his expanded class of functions. As a matter of fact, every function that can be computed by a Turing machine, can be represented by using the constructions above and vice versa, every construction can be represented by a Turing machine.

If you're intrigued, you could try to make Gödel's class bigger. You can try to define the 'opposite' of unbounded minimisation. That is, **unbounded maximisation** i.e. the function that finds the biggest root. However, you may find that computing that function is hard (impossible). You can read into the **Busy Beaver Problem**, which *tries* to apply unbounded maximisation.