# *THEORY OF COMPUTATION*
# *LECTURE NOTES*

**(Subject Code: BCS-303)**

*for*

*Bachelor of Technology*

*in*

*Computer Science and Engineering*

*&*

*Information Technology*


***Department of Computer Science and Engineering & Information Technology***

**Veer Surendra Sai University of Technology**

**(Formerly UCE, Burla)**

**Burla, Sambalpur, Odisha**

**Lecture Note Prepared by:** **Prof. D. Chandrasekhar Rao**

**Prof. Kishore Kumar Sahu**

**Prof. Pradipta Kumar Das**

# DISCLAIMER

This document does not claim any originality and cannot be used as a substitute for prescribed textbooks. The information presented here is merely a collection by the committee members for their respective teaching assignments. Various sources as mentioned at the end of the document as well as freely available material from internet were consulted for preparing this document. The ownership of the information lies with the respective authors or institutions.

**BCS 303** **THEORY OF COMPUTATION (3-1-0)** **Cr.-4**

**Module – I** **(10 Lectures)**

**Introduction to Automata:** The Methods Introduction to Finite Automata, Structural Representations, Automata and Complexity. Proving Equivalences about Sets, The Contrapositive, Proof by Contradiction, Inductive Proofs: General Concepts of Automata Theory: Alphabets Strings, Languages, Applications of Automata Theory.

**Finite Automata:** The Ground Rules, The Protocol, Deterministic Finite Automata: Definition of a Deterministic Finite Automata, How a DFA Processes Strings, Simpler Notations for DFA's, Extending the Transition Function to Strings, The Language of a DFA

Nondeterministic Finite Automata: An Informal View. The Extended Transition Function, The Languages of an NFA, Equivalence of Deterministic and Nondeterministic Finite Automata.

Finite Automata With Epsilon-Transitions: Uses of $\in$-Transitions, The Formal Notation for an $\in$-NFA, Epsilon-Closures, Extended Transitions and Languages for $\in$-NFA's, Eliminating $\in$-Transitions.

**Module – II** **(10 Lectures)**

**Regular Expressions and Languages:** Regular Expressions: The Operators of regular Expressions, Building Regular Expressions, Precedence of Regular-Expression Operators, Precedence of Regular-Expression Operators

Finite Automata and Regular Expressions: From DFA's to Regular Expressions, Converting DFA's to Regular Expressions, Converting DFA's to Regular Expressions by Eliminating States, Converting Regular Expressions to Automata.

Algebraic Laws for Regular Expressions:

**Properties of Regular Languages:** The Pumping Lemma for Regular Languages, Applications of the Pumping Lemma Closure Properties of Regular Languages, Decision Properties of Regular Languages, Equivalence and Minimization of Automata,

**Context-Free Grammars and Languages:** Definition of Context-Free Grammars, Derivations Using a Grammars Leftmost and Rightmost Derivations, The Languages of a Grammar,

**Parse Trees:** Constructing Parse Trees, The Yield of a Parse Tree, Inference Derivations, and Parse Trees, From Inferences to Trees, From Trees to Derivations, From Derivation to Recursive Inferences,

**Applications of Context-Free Grammars:** Parsers, Ambiguity in Grammars and Languages: Ambiguous Grammars, Removing Ambiguity From Grammars, Leftmost Derivations as a Way to Express Ambiguity, Inherent Anbiguity

**Module – III** **(10 Lectures)**

**Pushdown Automata:** Definition Formal Definition of Pushdown Automata, A Graphical Notation for PDA's, Instantaneous Descriptions of a PDA,

**Languages of PDA:** Acceptance by Final State, Acceptance by Empty Stack, From Empty Stack to Final State, From Final State to Empty Stack

Equivalence of PDA's and CFG's: From Grammars to Pushdown Automata, From PDA's to Grammars

**Deterministic Pushdown Automata:** Definition of a Deterministic PDA, Regular Languages and Deterministic PDA's, DPDA's and Context-Free Languages, DPDA's and Ambiguous Grammars

**Properties of Context-Free Languages:** Normal Forms for Context-Free Grammars, The Pumping Lemma for Context-Free Languages, Closure Properties of Context-Free Languages, Decision Properties of CFL's

**Module –IV** **(10 Lectures)**

**Introduction to Turing Machines:** The Turing Machine: The Instantaneous Descriptions for Turing Machines, Transition Diagrams for Turing Machines, The Language of a Turing Machine, Turing Machines and Halting

Programming Techniques for Turing Machines, Extensions to the Basic Turing Machine, Restricted Turing Machines, Turing Machines and Computers,

**Undecidability:** A Language That is Not Recursively Enumerable, Enumerating the Binary Strings, Codes for Turing Machines, The Diagonalization Language

An Undecidable Problem That Is RE: Recursive Languages, Complements of Recursive and RE languages, The Universal Languages, Undecidability of the Universal Language

Undecidable Problems About Turing Machines: Reductions, Turing Machines That Accept the Empty Language. Post's Correspondence Problem: Definition of Post's Correspondence Problem, The "Modified" PCP, Other Undecidable Problems: Undecidability of Ambiguity for CFG's

**Text Book:**


1. Introduction to Automata Theory Languages, and Computation, by J.E.Hopcroft, R.Motwani & J.D.Ullman (3rd Edition) – Pearson Education
2. Theory of Computer Science (Automata Language & Computations), by K.L.Mishra & N. Chandrashekhar, PHI

**MODULE-I**

**What is TOC?**

In theoretical computer science, the **theory of computation** is the branch that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches: automata theory, computability theory and computational complexity theory.

In order to perform a rigorous study of computation, computer scientists work with a mathematical abstraction of computers called a model of computation. There are several models in use, but the most commonly examined is the Turing machine.

Automata theory

In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata.

This automaton consists of

- **states** (represented in the figure by circles),
- and **transitions** (represented by arrows).

As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its ***transition function*** (which takes the current state and the recent symbol as its inputs).

Uses of Automata: compiler design and parsing.

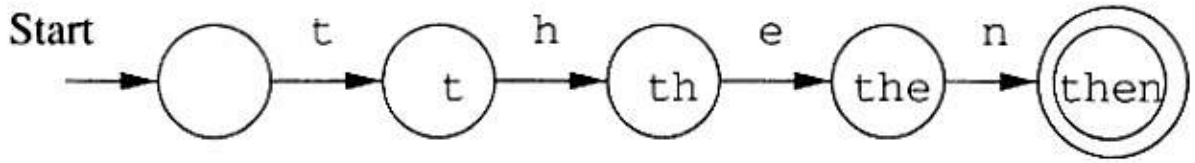



Figure 1.2: A finite automaton modeling recognition of **then**

**Introduction to formal proof:**

**Basic Symbols used :**

U – Union

∩- Conjunction

 - Empty String

Φ – NULL set

**7**- negation

**‘** – compliment

= > implies

**Additive inverse:** a+(-a)=0
**Multiplicative inverse:** a*1/a=1
Universal set U={1,2,3,4,5}
Subset A={1,3}
A' ={2,4,5}
**Absorption law:** AU(A ∩B) = A, A∩(AUB) = A

**De Morgan's Law:**
(AUB)' =A' ∩ B'
(A∩B)' = A' U B'
Double compliment
(A')' =A
A ∩ A' = Φ

**Logic relations:**
a → b = > **7**a U b
**7**(a∩b)=**7**a U **7**b

**Relations:**
Let a and b be two sets a relation R contains aXb.
Relations used in TOC:
**Reflexive:** a = a
**Symmetric:** aRb = > bRa
**Transition:** aRb, bRc = > aRc
If a given relation is reflexive, symmentric and transitive then the relation is called equivalence relation.

**Deductive proof:** Consists of sequence of statements whose truth lead us from some initial statement called the hypothesis or the give statement to a conclusion statement.

The theorem that is proved when we go from a hypothesis $H$ to a conclusion $C$ is the statement "if $H$ then $C$." We say that $C$ is *deduced* from $H$.

**Additional forms of proof:**
Proof of sets
Proof by contradiction
Proof by counter example

**Direct proof (AKA) Constructive proof:**
If *p* is true then *q* is true
Eg: if a and b are odd numbers then product is also an odd number.
Odd number can be represented as 2n+1
a=2x+1, b=2y+1
product of a X b = (2x+1) X (2y+1)
= 2(2xy+x+y)+1 = 2z+1 (odd number)

**Proof by contrapositive:**

The *contrapositive* of the statement "if $H$ then $C$" is "if not $C$ then not $H$." A statement and its contrapositive are either both true or both false, so we can prove either to prove the other.

**Theorem 1.10:** $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

| | Statement | Justification |
|---|---|---|
| 1. | $x$ is in $R \cup (S \cap T)$ | Given |
| 2. | $x$ is in $R$ or $x$ is in $S \cap T$ | (1) and definition of union |
| 3. | $x$ is in $R$ or $x$ is in both $S$ and $T$ | (2) and definition of intersection |
| 4. | $x$ is in $R \cup S$ | (3) and definition of union |
| 5. | $x$ is in $R \cup T$ | (3) and definition of union |
| 6. | $x$ is in $(R \cup S) \cap (R \cup T)$ | (4), (5), and definition of intersection |

Figure 1.5: Steps in the "if" part of Theorem 1.10

| | Statement | Justification |
|---|---|---|
| 1. | $x$ is in $(R \cup S) \cap (R \cup T)$ | Given |
| 2. | $x$ is in $R \cup S$ | (1) and definition of intersection |
| 3. | $x$ is in $R \cup T$ | (1) and definition of intersection |
| 4. | $x$ is in $R$ or $x$ is in both $S$ and $T$ | (2), (3), and reasoning about unions |
| 5. | $x$ is in $R$ or $x$ is in $S \cap T$ | (4) and definition of intersection |
| 6. | $x$ is in $R \cup (S \cap T)$ | (5) and definition of union |

Figure 1.6: Steps in the "only-if" part of Theorem 1.10

To see why "if $H$ then $C$" and "if not $C$ then not $H$" are logically equivalent, first observe that there are four cases to consider:

1. $H$ and $C$ both true.

2. $H$ true and $C$ false.

3. $C$ true and $H$ false.

4. $H$ and $C$ both false.

**Proof by Contradiction:**

H and not C implies falsehood.

That is, start by assuming both the hypothesis $H$ and the negation of the conclusion $C$. Complete the proof by showing that something known to be false follows logically from $H$ and not $C$. This form of proof is called *proof by contradiction*.

It often is easier to prove that a statement is not a theorem than to prove it *is* a theorem. As we mentioned, if $S$ is any statement, then the statement "$S$ is not a theorem" is itself a statement without parameters, and thus can

Be regarded as an observation than a theorem.

**Alleged Theorem 1.13:** All primes are odd. (More formally, we might say: if integer $x$ is a prime, then $x$ is odd.)

**DISPROOF**: The integer 2 is a prime, but 2 is even. □

For any sets a,b,c if a∩b = Φ and c is a subset of b the prove that a∩c =Φ
Given : a∩b=Φ and c subset b
Assume: a∩c ≠ Φ
Then $\forall x.\ x \in a \text{ and } x \in c \Rightarrow x \in b$
= > a∩b ≠Φ = > a∩c=Φ(i.e., the assumption is wrong)

**Proof by mathematical Induction:**

Suppose we are given a statement $S(n)$, about an integer $n$, to prove. One common approach is to prove two things:

1. The *basis*, where we show $S(i)$ for a particular integer $i$. Usually, $i = 0$ or $i = 1$, but there are examples where we want to start at some higher $i$, perhaps because the statement $S$ is false for a few small integers.
2. The *inductive step*, where we assume $n \geq i$, where $i$ is the basis integer, and we show that "if $S(n)$ then $S(n+1)$."

- ***The Induction Principle***: If we prove $S(i)$ and we prove that for all $n \geq i$, $S(n)$ implies $S(n+1)$, then we may conclude $S(n)$ for all $n \geq i$.

**Languages :**

The languages we consider for our discussion is an abstraction of natural languages. That is, our focus here is on formal languages that need precise and formal definitions. Programming languages belong to this category.

**Symbols :**

Symbols are indivisible objects or entity that cannot be defined. That is, symbols are the atoms of the world of languages. A symbol is any single object such as ♠, *a*, 0, 1, #, begin, or do.

**Alphabets :**

An alphabet is a finite, nonempty set of symbols. The alphabet of a language is normally denoted by $\Sigma$. When more than one alphabets are considered for discussion, then subscripts may be used (e.g. $\Sigma_1, \Sigma_2$ etc) or sometimes other symbol like G may also be introduced.

**Example** :

$\Sigma = \{0, 1\}$

$\Sigma = \{a, b, c\}$

$\Sigma = \{a, b, c, \&, z\}$

$\Sigma = \{\#, \nabla, \spadesuit, \beta\}$

**Strings or Words over Alphabet :**

A string or word over an alphabet $\Sigma$ is a finite sequence of concatenated symbols of $\Sigma$.

**Example** : 0110, 11, 001 are three strings over the binary alphabet { 0, 1 } .

aab, abcb, b, cc are four strings over the alphabet { a, b, c }.

It is not the case that a string over some alphabet should contain all the symbols from the alphabet. For example, the string cc over the alphabet { a, b, c } does not contain the symbols a and b. Hence, it is true that a string over an alphabet is also a string over any superset of that alphabet.

**Length of a string :**
The number of symbols in a string w is called its length, denoted by |w|.

**Example :** | 011 | = 4, |11| = 2, | b | = 1

**Convention :** We will use small case letters towards the beginning of the English alphabet to denote symbols of an alphabet and small case letters towards the end to denote strings over an alphabet. That is, $a, b, c \in \Sigma$ (symbols) and $u, v, w, x, y, z$ are strings.

**Some String Operations :**
Let $x = a_1a_2a_3 \in a_n$ and $y = b_1b_2b_3 \in b_m$ be two strings. The concatenation of x and y denoted by xy, is the string $a_1a_2a_3 \cdots a_nb_1b_2b_3 \cdots b_m$. That is, the concatenation of x and y denoted by xy is the string that has a copy of x followed by a copy of y without any intervening space between them.

**Example :** Consider the string 011 over the binary alphabet. All the prefixes, suffixes and substrings of this string are listed below.

Prefixes: $\varepsilon$, 0, 01, 011.
Suffixes: $\varepsilon$, 1, 11, 011.
Substrings: $\varepsilon$, 0, 1, 01, 11, 011.

Note that x is a prefix (suffix or substring) to x, for any string x and $\varepsilon$ is a prefix (suffix or substring) to any string.

A string x is a proper prefix (suffix) of string y if x is a prefix (suffix) of y and $x \neq y$.

In the above example, all prefixes except 011 are proper prefixes.

**Powers of Strings :** For any string x and integer $n \geq 0$, we use $x^n$ to denote the string formed by sequentially concatenating n copies of x. We can also give an inductive definition of $x^n$ as follows:
$x^n$ = e, if n = 0 ; otherwise $x^n = xx^{n-1}$

**Example :** If $x = 011$, then $x^3 = 011011011$, $x^1 = 011$ and $x^0 = e$

**Powers of Alphabets :**

We write $\Sigma^k$ (for some integer k) to denote the set of strings of length k with symbols from $\Sigma$. In other words,

$\Sigma^k = \{ w \mid w$ is a string over $\Sigma$ and $| w | = k \}$. Hence, for any alphabet, $\Sigma^0$ denotes the set of all strings of length zero. That is, $\Sigma^0 = \{ e \}$. For the binary alphabet { 0, 1 } we have the following.

$\Sigma^0 = \{e\}$.

$\Sigma^1 = \{0, 1\}$.

$\Sigma^2 = \{00, 01, 10, 11\}$.

$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

The set of all strings over an alphabet $\Sigma$ is denoted by $\Sigma^*$. That is,

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \Sigma^n \cup \cdots$$
$$= \cup \Sigma^k$$

The set $\Sigma^*$ contains all the strings that can be generated by iteratively concatenating symbols from $\Sigma$ any number of times.

**Example :** If $\Sigma = \{ a, b \}$, then $\Sigma^* = \{ \varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, \ldots \}$.

Please note that if $\Sigma = F$, then $\Sigma^*$ that is $\phi^* = \{e\}$. It may look odd that one can proceed from the empty set to a non-empty set by iterated concatenation. But there is a reason for this and we accept this convention

The set of all nonempty strings over an alphabet $\Sigma$ is denoted by $\Sigma^+$ That is,

$$\Sigma^+ = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \Sigma^n \cup \cdots$$
$$= \cup \Sigma^k$$

Note that $\Sigma^*$ is infinite. It contains no infinite strings but strings of arbitrary lengths.

**Reversal :**

For any string $w = a_1 a_2 a_3 \cdots a_n$ the reversal of the string is $w^R = a_n a_{n-1} \cdots a_3 a_2 a_1$.

An inductive definition of reversal can be given as follows:

**Languages :**
A language over an alphabet is a set of strings over that alphabet. Therefore, a language L is any subset of $\Sigma^*$. That is, any $L \subseteq \Sigma^*$ is a language.

**Example :**

1. F is the empty language.
2. $\Sigma^*$ is a language for any $\Sigma$.
3. {e} is a language for any $\Sigma$. Note that, $\phi \neq \{e\}$. Because the language F does not contain any string but {e} contains one string of length zero.
4. The set of all strings over { 0, 1 } containing equal number of 0's and 1's.
5. The set of all strings over {a, b, c} that starts with a.

**Convention :** Capital letters A, B, C, L, etc. with or without subscripts are normally used to denote languages.

**Set operations on languages :** Since languages are set of strings we can apply set operations to languages. Here are some simple examples (though there is nothing new in it).

**Union :** A string $x \in L_1 \cup L_2$ iff $x \in L_1$ or $x \in L_2$

**Example :** { 0, 11, 01, 011 } ∪{ 1, 01, 110 } = { 0, 11, 01, 011, 111 }

**Intersection :** A string, x∈ $L_1 \cap L_2$ iff x ∈ $L_1$ and x ∈ $L_2$ .

**Example :** { 0, 11, 01, 011 } ∩{ 1, 01, 110 } = { 01 }

Complement : Usually, $\Sigma^*$ is the universe that a complement is taken with respect to. Thus for a language L, the complement is L(bar) = { $x \in \Sigma^*$ | $x \notin L$ }.

**Example :** Let L = { x | |x| is even }. Then its complement is the language { $x \in \Sigma^*$ | |x| is odd }.
Similarly we can define other usual set operations on languages like relative complement, symmetric difference, etc.

**Reversal of a language :**
The reversal of a language $L$, denoted as $L^R$, is defined as: $L^R = \{w^R \mid w \in L\}$.

**Example :**

1. Let L = { 0, 11, 01, 011 }. Then $L^R$ = { 0, 11, 10, 110 }.

2. Let L = { $1^n0^n$ | n is an integer }. Then $L^R$ = { $1^n0^n$ | n is an integer }.

**Language concatenation :** The concatenation of languages $L_1$ and $L_2$ is defined as $L_1 L_2$ = { $xy$ | $x \in L_1$ and $y \in L_2$ }.

**Example :** { *a*, *ab* }{ *b*, *ba* } = { *ab*, *aba*, *abb*, *abba* }.

Note that ,

1. $L_1 L_2 \neq L_2 L_1$ in general.
2. $L\Phi = \Phi$
3. $L\{\varepsilon\} = L = \{\varepsilon\}$

**lterated concatenation of languages :** Since we can concatenate two languages, we also repeat this to concatenate any number of languages. Or we can concatenate a language with itself any number of times. The operation $L^n$ denotes the concatenation of L with itself n times. This is defined formally as follows:

$L_0 = \{e\}$
$L^n = LL^{n-1}$

**Example :** Let L = { a, ab }. Then according to the definition, we have

$L_0 = \{e\}$
$L_1 = L\{e\} = L = \{a, ab\}$
$L_2 = L\,L_1 = \{a, ab\}\{a, ab\} = \{aa, aab, aba, abab\}$
$L_3 = L\,L_2 = \{a, ab\}\{aa, aab, aba, abab\}$
$= \{aaa, aaab, aaba, aabab, abaa, abaab, ababa, ababab\}$

and so on.

**Kleene's Star operation :** The Kleene star operation on a language L, denoted as $L^*$ is defined as follows :

$L^*$ = ( Union n in N ) $L^n$

$= L^0 \cup L^1 \cup L^2 \cup \cdots$

= { x | x is the concatenation of zero or more strings from L }

Thus $L^*$ is the set of all strings derivable by any number of concatenations of strings in L. It is also useful to define

$L^+ = LL^*$, i.e., all strings derivable by one or more concatenations of strings in L. That is

$L^+$ = (Union n in N and n >0) $L^n$

$= L^1 \cup L^2 \cup L^3 \cup \cdots$

**Example :** Let L = { a, ab }. Then we have,

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

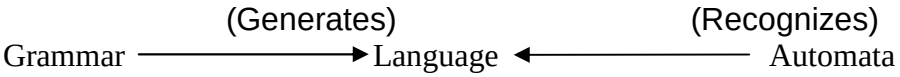
$= \{e\} \cup \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots$

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

$= \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots$

Note : $\varepsilon$ is in $L^*$, for every language L, including .

The previously introduced definition of $\Sigma^*$ is an instance of Kleene star.




Automata: A algorithm or program that automatically recognizes if a particular string belongs to the language or not, by checking the grammar of the string.

An automata is an abstract computing device (or machine). There are different varities of such abstract machines (also called models of computation) which can be defined mathematically.

Every Automaton fulfills the three basic requirements.

- Every automaton consists of some essential features as in real computers. It has a mechanism for reading input. The input is assumed to be a sequence of symbols over a given alphabet and is placed on an input tape(or written on an input file). The simpler automata can only read the input one symbol at a time from left to right but not change. Powerful versions can both read (from left to right or right to left) and change the input.

- The automaton can produce output of some form. If the output in response to an input string is binary (say, accept or reject), then it is called an accepter. If it produces an output sequence in response to an input sequence, then it is called a transducer(or automaton with output).
- The automaton may have a temporary storage, consisting of an unlimited number of cells, each capable of holding a symbol from an alphabet ( whcih may be different from the input alphabet). The automaton can both read and change the contents of the storage cells in the temporary storage. The accusing capability of this storage varies depending on the type of the storage.
- The most important feature of the automaton is its control unit, which can be in any one of a finite number of interval states at any point. It can change state in some defined manner determined by a transition function.

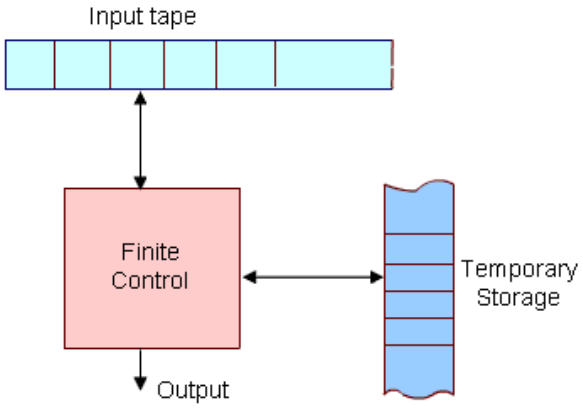



Figure 1: The figure above shows a diagrammatic representation of a generic automation.

Operation of the automation is defined as follows.
At any point of time the automaton is in some integral state and is reading a particular symbol from the input tape by using the mechanism for reading input. In the next time step the automaton then moves to some other integral (or remain in the same state) as defined by the transition function. The transition function is based on the current state, input symbol read, and the content of the temporary storage. At the same time the content of the storage may be changed and the input read may be modifed. The automation may also produce some output during this transition. The internal state, input and the content of storage at any point defines the configuration of the automaton at that point. The transition from one configuration to the next ( as defined by the transition function) is called a *move.* Finite state machine or *Finite Automation* is the simplest type of abstract machine we consider. Any system that is at any point of time in one of a finite number of interval state and moves among these states in a defined manner in response to some input, can be modeled by a finite automaton. It doesnot have any temporary storage and hence a restricted model of computation.

**Finite Automata**

Automata (singular : automation) are a particularly simple, but useful, model of computation. They were initially proposed as a simple model for the behavior of neurons.

**States, Transitions and Finite-State Transition System :**

Let us first give some intuitive idea about *a state of a system* and *state transitions* before describing finite automata.

Informally, *a state of a system* is an instantaneous description of that system which gives all relevant information necessary to determine how the system can evolve from that point on.

*Transitions* are changes of states that can occur spontaneously or in response to inputs to the states. Though transitions usually take time, we assume that state transitions are instantaneous (which is an abstraction).

Some examples of state transition systems are: digital systems, vending machines, etc. A system

containing only a finite number of states and transitions among them is called a *finite-state transition system*.

Finite-state transition systems can be modeled abstractly by a mathematical model called *finite automation*

**Deterministic Finite (-state) Automata**

Informally, a DFA (Deterministic Finite State Automaton) is a simple machine that reads an input string -- one symbol at a time -- and then, after the input has been completely read, decides whether to accept or reject the input. As the symbols are read from the tape, the automaton can change its state, to reflect how it reacts to what it has seen so far. A machine for which a deterministic code can be formulated, and if there is only one unique way to formulate the code, then the machine is called deterministic finite automata.

Thus, a DFA conceptually consists of 3 parts:

1. A *tape* to hold the input string. The tape is divided into a finite number of cells. Each cell holds a symbol from $\Sigma$.
2. A *tape head* for reading symbols from the tape
3. A *control* , which itself consists of 3 things:
    - finite number of states that the machine is allowed to be in (zero or more states are designated as *accept* or *final* states),
    - a current state, initially set to a start state,

- a state transition function for changing the current state.

An automaton processes a string on the tape by repeating the following actions until the tape head has traversed the entire string:

1. The tape head reads the current tape cell and sends the symbol s found there to the control. Then the tape head moves to the next cell.
2. he control takes s and the current state and consults the state transition function to get the next state, which becomes the new current state.

Once the entire string has been processed, the state in which the automation enters is examined.

If it is an accept state , the input string is accepted ; otherwise, the string is rejected . Summariz-

ing all the above we can formulate the following formal definition:

**Deterministic Finite State Automaton :** A Deterministic Finite State Automaton (DFA) is a 5-tuple : $M = (Q, \Sigma, \delta, q_0, F)$

- $Q$ is a finite set of states.
- $\Sigma$ is a finite set of input symbols or alphabet
- $\delta : Q \times \Sigma \to Q$ is the "next state" transition function (which is total ). Intuitively, is $\delta$ a function that tells which state to move to in response to an input, i.e., if M is in state q and sees input a, it moves to state $\delta(q,a)$.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accept or final states.

**Acceptance of Strings :**

A DFA accepts a string $w = a_1 a_2 \cdots a_n$ if there is a sequence of states $q_0, q_1, \cdots, q_n$ in $Q$ such that

1. $q_0$ is the start state.
2. $\delta(q_i, q_{i+1}) = a_{i+1}$ for all $0 < i < n$.
3. $q_n \in F$

**Language Accepted or Recognized by a DFA :**

The language accepted or recognized by a DFA M is the set of all strings accepted by M , and is denoted by $L(M)$ i.e. $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$. The notion of acceptance can also be made more precise by extending the transition function $\delta$.

**Extended transition function :**

Extend $\delta : Q \times \Sigma \to Q$ (which is function on symbols) to a function on strings, i.e. . $\delta^{\wedge} : Q \times \Sigma^{*} \to Q$

That is, $\delta^{\wedge}(q, w)$ is the state the automation reaches when it starts from the state q and finish processing the string w. Formally, we can give an inductive definition as follows:
The language of the DFA M is the set of strings that can take the start state to one of the accepting states i.e.

$$L(M) = \{ w \in \Sigma^{*} \mid M \text{ accepts } w \}$$

$$= \{ w \in \Sigma^{*} \mid \delta^{\wedge}(q_0, w) \in F \}$$

**Example 1 :**

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

$Q = \{q_0, q_1\}$

$q_0$ is the start state

$F = \{q_1\}$

$\delta(q_0, 0) = q_0 \qquad \delta(q_1, 0) = q_1$

$\delta(q_0, 1) = q_1 \qquad \delta(q_1, 1) = q_1$

It is a formal description of a DFA. But it is hard to comprehend. For ex. The language of the DFA is any string over { 0, 1} having at least one 1

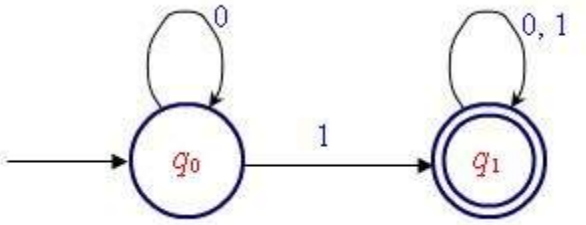
We can describe the same DFA by transition table or state transition diagram as following:

**Transition Table :**

| | 0 | 1 |
|---|---|---|
| $\to q_0$ | $q_0$ | $q_1$ |

| | | |
|---|---|---|
| $*q_1$ | $q_1$ | $q_1$ |

It is easy to comprehend the transition diagram.




**Explanation :** We cannot reach find state $q_1$ w/0 or in the i/p string. There can be any no. of 0's at the beginning. ( The self-loop at $q_0$ on label 0 indicates it ). Similarly there can be any no. of 0's & 1's in any order at the end of the string.

**Transition table :**

It is basically a tabular representation of the transition function that takes two arguments (a state and a symbol) and returns a value (the "next state").

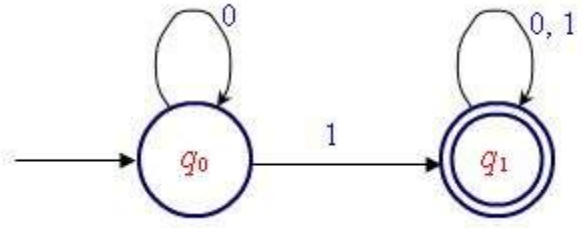
- Rows correspond to states,
- Columns correspond to input symbols,
- Entries correspond to next states
- The start state is marked with an arrow
- The accept states are marked with a star (*).

| | 0 | 1 |
|---|---|---|
| $\rightarrow q_0$ | $q_0$ | $q_1$ |
| $*q_1$ | $q_1$ | $q_1$ |

**(State) Transition diagram :**

A state transition diagram or simply a transition diagram is a directed graph which can be constructed as follows:

1. For each state in Q there is a node.
2. There is a directed edge from node q to node p labeled a iff $\delta(q, a) = p$ . (If there are several input symbols that cause a transition, the edge is labeled by the list of these symbols.)
3. There is an arrow with no source into the start state.
4. Accepting states are indicated by double circle.

5. 


6. Here is an informal description how a DFA operates. An input to a DFA can be any string $w \in \Sigma^*$ Put a pointer to the start state q. Read the input string w from left to right, one symbol at a time, moving the pointer according to the transition function, $\delta$. If the next symbol of w is a and the pointer is on state p, move the pointer to $\delta(p, a)$. When the end of the input string w is encountered, the pointer is on some state, r. The string is said to be accepted by the DFA if $r \in F$ and rejected if $r \notin F$. Note that there is no formal mechanism for moving the pointer.
7. A language $L \in \Sigma^*$ is said to be regular if L = L(M) for some DFA M.

**Regular Expressions: Formal Definition**

We construct REs from primitive constituents (basic elements) by repeatedly applying certain recursive rules as given below. (In the definition)

**Definition** : Let S be an alphabet. The regular expressions are defined recursively as follows.

**Basis** :

i) $\phi$ is a RE

ii) $\in$ is a RE

iii) $\forall\, a \in S$, a is RE.

These are called primitive regular expression i.e. Primitive Constituents

**Recursive Step :**

If $r_1$ and $r_2$ are REs over, then so are

i) $r_1 + r_2$

ii) $r_1 r_2$

# MODULE-II

## Regular Expressions: Formal Definition

We construct REs from primitive constituents (basic elements) by repeatedly applying certain recursive rules as given below. (In the definition)

**Definition :** Let $S$ be an alphabet. The regular expressions are defined recursively as follows.

**Basis :**

i) $\phi$ is a RE

ii) $\in$ is a RE

iii) $\forall\ a \in S$ , $a$ is RE.

These are called primitive regular expression i.e. Primitive Constituents

**Recursive Step :**

If $r_1$ and $r_2$ are REs over, then so are

i) $r_1 + r_2$

ii) $r_1 r_2$

iii) $r_1^*$

iv) $(r_1)$

**Closure :** *r* is RE over only if it can be obtained from the basis elements (Primitive REs) by a finite no of applications of the recursive step (given in 2).

**Example :** Let $\Sigma$ = { 0,1,2 }. Then (0+21)*(1+ $F$ ) is a RE, because we can construct this expression by applying the above rules as given in the following step.

| Steps | RE Constructed | Rule Used |
|---|---|---|
| 1 | 1 | Rule 1(iii) |
| 2 | $\phi$ | Rule 1(i) |
| 3 | $1+\phi$ | Rule 2(i) & Results of Step 1, 2 |

| | | |
|---|---|---|
| 4 | (1+$\phi$) | Rule 2(iv) & Step 3 |
| 5 | 2 | 1(iii) |
| 6 | 1 | 1(iii) |
| 7 | 21 | 2(ii), 5, 6 |
| 8 | 0 | 1(iii) |
| 9 | 0+21 | 2(i), 7, 8 |
| 10 | (0+21) | 2(iv), 9 |
| 11 | (0+21)* | 2(iii), 10 |
| 12 | (0+21)* | 2(ii), 4, 11 |

**Language described by REs :** Each describes a language (or a language is associated with every RE). We will see later that REs are used to attribute regular languages.

**Notation :** If *r* is a RE over some alphabet then $L(r)$ is the language associate with *r* . We can define the language $L(r)$ associated with (or described by) a REs as follows.

1. $\phi$ is the RE describing the empty language i.e. $L(\phi) = \phi$.

2. $\in$ is a RE describing the language {$\in$} i.e. $L(\in) = \{\in\}$ .

3. $\forall\ a\ \in S$ , *a* is a RE denoting the language {*a*} i.e . $L$(a) = {a} .

4. If $r_1$ and $r_2$ are REs denoting language $L(r_1)$ and $L(r_2)$ respectively, then

i) $r_1 + r_2$ is a regular expression denoting the language $L(r_1 + r_2) = L(r_1) \cup L(r_2)$

ii) $r_1 r_2$ is a regular expression denoting the language $L(r_1 r_2) = L(r_1)\ L(r_2)$

iii) $r_1^*$ is a regular expression denoting the language $L(r_1^*) = (L(r1))^*$

iv) ($r_1$) is a regular expression denoting the language $L((r_1)) = L(r_1)$

**Example :** Consider the RE (0*(0+1)). Thus the language denoted by the RE is

$L$(0*(0+1)) = $L$(0*) $L$(0+1) .....by 4(ii)

= $L$(0)*$L$(0) $\cup$ $L$(1)

= {$\in$ , 0,00,000,.....} {0} $\cup$ {1}

= {$\in$ , 0,00,000,.....} {0,1}

= {0, 00, 000, 0000,.....,1, 01, 001, 0001,.....}

**Precedence Rule**

Consider the RE $ab + c$. The language described by the RE can be thought of either $L(a)L(b+c)$ or $L(ab) \cup L(c)$ as provided by the rules (of languages described by REs) given already. But these two represents two different languages lending to ambiguity. To remove this ambiguity we can either

1) Use fully parenthesized expression- (cumbersome) or

2) Use a set of precedence rules to evaluate the options of REs in some order. Like other algebras mod in mathematics.

For REs, the order of precedence for the operators is as follows:

i) The star operator precedes concatenation and concatenation precedes union (+) operator.

ii) It is also important to note that concatenation & union (+) operators are associative and union operation is commutative.

Using these precedence rule, we find that the RE ab+c represents the language $L(ab) \cup L(c)$ i.e. it should be grouped as $((ab)+c)$.

We can, of course change the order of precedence by using parentheses. For example, the language represented by the RE $a(b+c)$ is $L(a)L(b+c)$.

**Example :** The RE $ab^*+b$ is grouped as $((a(b^*))+b)$ which describes the language $L(a)(L(b))^* \cup L(b)$

**Example :** The RE $(ab)^*+b$ represents the language $(L(a)L(b))^* \cup L(b)$.

**Example :** It is easy to see that the RE (0+1)*(0+11) represents the language of all strings over {0,1} which are either ended with 0 or 11.

**Example :** The regular expression $r$ =(00)*(11)*1 denotes the set of all strings with an even number of 0's followed by an odd number of 1's i.e. $L(r) = \{0^{2n}1^{2m+1} \mid n \geq 0,\ m \geq 0\}$

**Note :** The notation $r^+$ is used to represent the RE $rr^*$. Similarly, $r^2$ represents the RE $rr$, $r^3$ denotes $r^2 r$, and so on.

An arbitrary string over $\Sigma$ = {0,1} is denoted as (0+1)*.

**Exercise :** Give a RE $r$ over {0,1} s.t. $L(r)$={$w \in \Sigma^* \mid w$ has at least one pair of consecutive 1's}

**Solution :** Every string in $L(r)$ must contain 00 somewhere, but what comes before and what goes before is completely arbitrary. Considering these observations we can write the REs as (0+1)*11(0+1)*.

**Example :** Considering the above example it becomes clean that the RE (0+1)*11(0+1)*+(0+1)*00(0+1)* represents the set of string over {0,1} that contains the substring 11 or 00.

**Example :** Consider the RE 0*10*10*. It is not difficult to see that this RE describes the set of strings over {0,1} that contains exactly two 1's. The presence of two 1's in the RE and any no of 0's before, between and after the 1's ensure it.

**Example :** Consider the language of strings over {0,1} containing two or more 1's.

**Solution :** There must be at least two 1's in the RE somewhere and what comes before, between, and after is completely arbitrary. Hence we can write the RE as (0+1)*1(0+1)*1(0+1)*. But following two REs also represent the same language, each ensuring presence of least two 1's somewhere in the string

i) 0*10*1(0+1)*

ii) (0+1)*10*10*

**Example :** Consider a RE $r$ over {0,1} such that

$L(r)$ = { $w \in (0,1)^* \mid w$ has no pair of consecutive 1's}

**Solution :** Though it looks similar to ex ……., it is harder to construct to construct. We observer that, whenever a 1 occurs, it must be immediately followed by a 0. This substring may be preceded & followed by any no of 0's. So the final RE must be a repetition of strings of the form: 00…0100….00 i.e. 0*100*. So it looks like the RE is (0*100*)*. But in this case the strings ending in 1 or consisting of all 0's are not accounted for. Taking these observations into consideration, the final RE is $r$ = (0*100*)(1+ $\in$)+0*(1+$\in$).

**Alternative Solution :**
The language can be viewed as repetitions of the strings 0 and 01. Hence get the RE as $r$ = (0+10)*(1+$\in$).This is a shorter expression but represents the same language.

## Regular Expression:

## FA to regular expressions:

### FA to RE (REs for Regular Languages) :

**Lemma :** If a language is regular, then there is a RE to describe it. i.e. if $L = L(M)$ for some DFA $M$, then there is a RE $r$ such that $L = L(r)$.

**Proof :** We need to construct a RE $r$ such that $L(r) = \{ w \mid w \in L(M) \}$ . Since $M$ is a DFA, it has a finite no of states. Let the set of states of $M$ is $Q$ = {1, 2, 3,..., $n$} for some integer n. [ Note : if the $n$ states of $M$ were denoted by some other symbols, we can always rename those to indicate as 1, 2, 3,..., $n$ ]. The required RE is constructed inductively.

**Notations :** $r_{ij}^{(k)}$ is a RE denoting the language which is the set of all strings $w$ such that $w$ is the label of a path from state $i$ to state $j$ $(1 \le i, j \le n)$ in $M$, and that path has no intermediate state whose number is greater then $k$. ( $i$ & $j$ (begining and end pts) are not considered to be "intermediate" so $i$ and /or $j$ can be

Let the given CFG is $G=(N,\Sigma,P,S)$. Without loss of generality we can assume that $G$ is in Greibach Normal Form i.e. all productions of $G$ are of the form .

$A\to cB_1B_2\cdots B_k$ where $c\in\Sigma\cup\{\varepsilon\}$ and $k\geq 0$ .

From the given CFG $G$ we now construct an equivalent PDA $M$ that accepts by empty stack. Note that there is only one state in $M$. Let

$M=(\{q\},\Sigma,N,\delta,q,S,\phi)$, where

- $q$ is the only state
- $\Sigma$ is the input alphabet,
- $N$ is the stack alphabet ,
- $q$ is the start state.
- $S$ is the start/initial stack symbol, and $\delta$, the transition relation is defined as follows

  For each production $A\to cB_1B_2\cdots B_k\in P$, $(q,B_1B_2\ldots B_k)\in\delta(q,c,A)$. We now want to show that $M$ and $G$ are equivalent i.e. $L(G)=N(M)$. i.e. for any $w\in\Sigma^*$. $w\in L(G)$ iff $w\in N(M)$.

If $w\in L(G)$, then by definition of $L(G)$, there must be a leftmost derivation starting with $S$ and deriving $w$.

i.e. $S\overset{*}{\underset{G}{\Rightarrow}}w$

Again if $w\in N(M)$, then one sysmbol. Therefore we need to show that for any $w\in\Sigma^*$.

$S\overset{*}{\underset{G}{\Rightarrow}}w$ iff $(q,w,s)\vdash(q,\in,\in)$.

But we will prove a more general result as given in the following lemma. Replacing $A$ by $S$ (the start symbol) and $\gamma$ by $\in$ gives the required proof.

Lemma For any $x,\ y\in\Sigma^*$, $\gamma\in N^*$ and $A\in N$, $A\overset{n}{\underset{G}{\Rightarrow}}x\gamma$ via a leftmost derivative iff $(q,xy,A)\vdash_M^n(q,y,\gamma)$.

**Proof :** The proof is by induction on $n$.

**Basis :** $n = 0$

$A \overset{0}{\underset{G}{\Rightarrow}} x\gamma$ iff $A = x\gamma$ i.e. $x = \in$ and $\gamma = A$

iff $(q, xy, A) = (q, y, \gamma)$

iff $(q, xy, A) \vdash_M^0 (q, y, \gamma)$

**Induction Step :**

First, assume that $A \overset{n+1}{\underset{G}{\Rightarrow}} x\gamma$ via a leftmost derivation. Let the last production applied in their derivation is $B \to c\beta$ for some $c \in \Sigma \cup \{\in\}$ and $\beta \in N^*$.

Then, for some $\omega \in \Sigma^*$, $\alpha \in N^*$

$A \overset{n}{\underset{G}{\Rightarrow}} \omega B\alpha \overset{1}{\underset{G}{\Rightarrow}} \omega c\,\beta\alpha = x\gamma$

where $x = \omega c$ and $\gamma = \beta\alpha$

Now by the indirection hypothesis, we get,

$(q, \omega cy, A) \vdash_M^n (q, cy, B\alpha)$ .....(1)

Again by the construction of $M$, we get

$(q, \beta) \in \delta(q, c, B)$

so, from (1), we get

$(q, \omega cy, A) \vdash_M^n (q, cy, B\alpha) \vdash_M^1 (q, y, \beta\alpha)$

since $x = \omega c$ and $\gamma = \beta\alpha$, we get $(q, xy, A) \vdash_M^{n+1} (q, y, \gamma)$

That is, if $A \overset{n+1}{\underset{G}{\Rightarrow}} x\gamma$, then $(q, xy, A) \vdash_M^{n+1} (q, y, \gamma)$. Conversely, assume that $(q, xy, A) \vdash_M^{n+1} (q, y, \gamma)$ and let

$\delta(q,c,B)=(q,\beta)$ be the transition used in the last move. Then for some $\varpi \in \Sigma^*$, $c \in \Sigma \cup \{\in\}$ and $\alpha \in \Gamma^*$

$(q,\varpi cy,A) \vdash_M^n (q,cy,B\alpha) \vdash_M^1 (q,y,\beta\alpha)$ where $x=\varpi c$ and $\gamma=\beta\alpha$.

Now, by the induction hypothesis, we get

$A \overset{n}{\underset{G}{\Rightarrow}} \varpi B\alpha$ via a leftmost derivation.

Again, by the construction of $M$, $B \to c\beta$ must be a production of $G$. [ Since $(q, \beta) \in \delta(q, c, B)$]. Applying the production to the sentential form $\varpi B\alpha$ we get

$A \overset{n}{\underset{G}{\Rightarrow}} \varpi B\alpha \overset{1}{\underset{G}{\Rightarrow}} \varpi c\beta\alpha = x\gamma$

i.e. $A \overset{n+1}{\underset{G}{\Rightarrow}} x\gamma$

via a leftmost derivation.

Hence the proof.

**Example :** Consider the CFG $G$ in GNF

S $\to$ aAB
A $\to$ a / aA
B $\to$ a / bB

The one state PDA $M$ equivalent to $G$ is shown below. For convenience, a production of $G$ and the corresponding transition in $M$ are marked by the same encircled number.

(1) $S \to aAB$
(2) $A \to a$
(3) $A \to aA$
(4) $B \to a$
(5) $B \to bB$

$M = (\{q\}, \{a,b\}, \{S,A,B\}, \delta, q, S, \Sigma)$. We have used the same construction discussed earlier

**Some Useful Explanations :**
Consider the moves of M on input *aaaba* leading to acceptance of the string.
Steps

1. $(q, aaaba, s) \overset{(1)}{\underset{M}{\vdash}} (q, aaba, AB)$

2. $\overset{(2)}{\underset{M}{\vdash}} (q, aba, AB)$

3. $\overset{(3)}{\underset{M}{\vdash}} (q, ba, B)$

4. $\overset{(4)}{\underset{M}{\vdash}} (q, a, B)$

5. $\overset{(5)}{\underset{M}{\vdash}} (q, \in, \in)$ Accept by empty stack.

**Note :** encircled numbers here shows the transitions rule applied at every step.
Now consider the derivation of the same string under grammar G. Once again, the production used at every step is shown with encircled number.

$$S \overset{(1)}{\underset{G}{\Rightarrow}} aAB \overset{(3)}{\underset{G}{\Rightarrow}} aaAB \overset{(2)}{\underset{G}{\Rightarrow}} aaaB \overset{(5)}{\underset{G}{\Rightarrow}} aaabB \overset{(4)}{\underset{G}{\Rightarrow}} aaaba$$

Steps $\rightarrow$ 1 $\rightarrow$ 2 $\rightarrow$ 3 $\rightarrow$ 4 $\rightarrow$ 5

Observations:

- There is an one-to-one correspondence of the sequence of moves of the PDA $M$ and the derivation sequence under the CFG $G$ for the same input string in the sense that - number of steps in both the cases are same and transition rule corresponding to the same production is used at every step (as shown by encircled number).
- considering the moves of the PDA and derivation under $G$ together, it is also observed that at every step the input read so far and the stack content together is exactly identical to the corresponding sentential form i.e.
  <what is Read><stack> = <sentential form>

Say, at step 2, Read so far = $a$
stack = $AB$

Sentential form = $aAB$ From this property we claim that $(q, x, S) \vdash_M^* (q, \in, \alpha)$ iff $S \overset{*}{\underset{G}{\Rightarrow}} x\alpha$. If the claim is true, then apply with $\alpha = \in$ and we get $(q, x, S) \vdash_M^* (q, \in, \in)$ iff $S \overset{*}{\underset{G}{\Rightarrow}} x$ or $x \in N(M)$ iff $x \in L(G)$ ( by definition )

Thus $N(M) = L(G)$ as desired. Note that we have already proved a more general version of the claim

PDA and CFG:

We now want to show that for every PDA $M$ that accpets by empty stack, there is a CFG $G$ such that $L(G) = N(M)$

we first see whether the "reverse of the construction" that was used in part (i) can be used here to construct an equivalent CFG from any PDA $M$.

It can be show that this reverse construction works only for single state PDAs.

**Empty Production Removal**

The productions of context-free grammars can be coerced into a variety of forms without affecting the expressive power of the grammars. If the empty string does not belong to a language, then there is a way to eliminate the productions of the form $A \rightarrow \lambda$ from the grammar.

If the empty string belongs to a language, then we can eliminate $\lambda$ from all productions save for the single production $S \rightarrow \lambda$. In this case we can also eliminate any occurrences of $S$ from the right-hand side of productions.

**Procedure to find CFG with out empty Productions**

*Step (i):* For all productions $A \rightarrow \lambda$, put $A$ into $V_N$.

*Step (ii):* Repeat the following steps until no further variables are added to $V_N$.
For all productions

$$B \rightarrow A_1 A_2 \ \ldots\ldots \ A_n .$$

*Step (i):* For all productions $A \rightarrow \lambda$, put $A$ into $V_N$.

*Step (ii):* Repeat the following steps until no further variables are added to $V_N$.
For all productions

$$B \rightarrow A_1 A_2 \ \ldots\ldots \ A_n .$$

where $A_1, A_2, A_3, \ldots\ldots, A_n$ are in $V_N$, put $B$ into $V_N$.

To find $\hat{P}$, let us consider all productions in $P$ of the form

$$A \to x_1 x_2 \ \ldots\ldots\ x_m, m \geq 1$$

for each $x_i \in V \cup T$.

**Unit production removal**

Any production of a CFG of the form

$$A \to B$$

where $A, B \in V$ is called a "Unit-production". Having variable one on either side of a production is sometimes undesirable.

"Substitution Rule" is made use of in removing the unit-productions.

Given $G = (V, T, S, P)$, a CFG with no $\lambda$-productions, there exists a CFG $\hat{G} = (\hat{V}, \hat{T}, S, \hat{P})$ that does not have any unit-productions and that is equivalent to $G$.

Let us illustrate the procedure to remove unit-production through example 2.4.6.

*Procedure to remove the unit productions:*

Find all variables $B$, for each $A$ such that

$$A \overset{*}{\Rightarrow} B$$

This is done by sketching a "depending graph" with an edge $(C, D)$ whenever the grammar has unit-production $C \to D$, then $A \overset{*}{\Rightarrow} B$ holds whenever there is a walk between $A$ and $B$.

The new grammar $\hat{G}$, equivalent to $G$ is obtained by letting into $\hat{P}$ all non-unit productions of $P$.

Then for all $A$ and $B$ satisfying $A \overset{*}{\Rightarrow} B$, we add to $\hat{P}$

$$A \to y_1 \mid y_2 \mid \ldots\ldots \mid y_n$$

where $B \to y_1 \mid y_2 \mid \ldots\ldots \mid y_n$ is the set of all rules in $\hat{P}$ with $B$ on the left.

**Left Recursion Removal**

A variable $A$ is left-recursive if it occurs in a production of the form

$$A \rightarrow Ax$$

for any $x \in (V \cup T)^*$.

A grammar is left-recursive if it contains at least one left-recursive variable.

Every content-free language can be represented by a grammar that is not left-recursive.

**NORMAL FORMS**

Two kinds of normal forms viz., Chomsky Normal Form and Greibach Normal Form (GNF) are considered here.

**Chomsky Normal Form (CNF)**

Any context-free language $L$ without any λ-production is generated by a grammar is which productions are of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B $\in V_N$, and a $\in V_T$.

**Procedure to find Equivalent Grammar in CNF**

(i) Eliminate the unit productions, and λ-productions if any,
(ii) Eliminate the terminals on the right hand side of length two or more.
(iii) Restrict the number of variables on the right hand side of productions to two.

Proof:

*For Step (i):* Apply the following theorem: "Every context free language can be generated by a grammar with no useless symbols and no unit productions".

At the end of this step the RHS of any production has a single terminal or two or more symbols. Let us assume the equivalent resulting grammar as $G = (V_N, V_T, P, S)$.

*For Step (ii):* Consider any production of the form

$$A \rightarrow y_1 y_2 \ldots\ldots y_m, \quad m \geq 2.$$

If $y_1$ is a terminal, say '$a$', then introduce a new variable $B_a$ and a production

$$B_a \rightarrow a$$

Repeat this for every terminal on RHS.

Let $P'$ be the set of productions in $P$ together with the new productions

$B_a \to a$. Let $V_N'$ be the set of variables in $V_N$ together with $B_a's$ introduced for every terminal on RHS.

The resulting grammar $G_1 = (V_N', V_T, P', S)$ is equivalent to $G$ and every production in $P'$ has either a single terminal or two or more variables.

*For step (iii):* Consider $A \to B_1 B_2 \ldots\ldots B_m$

where $B_i$'s are variables and $m \geq 3$.

If $m = 2$, then $A \to B_1, B_2$ is in proper form.

The production $A \to B_1 B_2 \ldots\ldots B_m$ is replaced by new productions

$$\begin{aligned} A &\to B_1 D_1, \\ D_1 &\to B_2 D_2, \\ &\ldots\ldots \ \ldots\ldots \\ &\ldots\ldots \ \ldots\ldots \\ D_{m-2} &\to B_{m-1} B_m \end{aligned}$$

where $D_i'S$ are new variables.

The grammar thus obtained is $G_2$, which is in CNF.

**Example**

Obtain a grammar in Chomsky Normal Form (CNF) equivalent to the grammar *G* with productions *P* given

$S \to aAbB$

$A \to aA \mid a$

$B \to bB \mid b.$

**Solution**

**Theorem 7.30:** Let $L$ be a CFL and $h$ a homomorphism. Then $h^{-1}(L)$ is a CFL.

**PROOF:** Suppose $h$ applies to symbols of alphabet $\Sigma$ and produces strings in $T^*$. We also assume that $L$ is a language over alphabet $T$. As suggested above, we start with a PDA $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$ that accepts $L$ by final state. We construct a new PDA

$$P' = (Q', \Sigma, \delta', (q_0, \epsilon), Z_0, F \times \{\epsilon\}) \tag{7.1}$$

where:

1. $Q'$ is the set of pairs $(q, x)$ such that:

   (a) $q$ is a state in $Q$, and

   (b) $x$ is a suffix (not necessarily proper) of some string $h(a)$ for some input symbol $a$ in $\Sigma$.

   That is, the first component of the state of $P'$ is the state of $P$, and the second component is the buffer. We assume that the buffer will periodically be loaded with a string $h(a)$, and then allowed to shrink from the front, as we use its symbols to feed the simulated PDA $P$. Note that since $\Sigma$ is finite, and $h(a)$ is finite for all $a$, there are only a finite number of states for $P'$.

2. $\delta'$ is defined by the following rules:

   (a) $\delta'\big((q, \epsilon), a, X\big) = \{\big((q, h(a)\big), X\big)\}$ for all symbols $a$ in $\Sigma$, all states $q$ in $Q$, and stack symbols $X$ in $\Gamma$. Note that $a$ cannot be $\epsilon$ here. When the buffer is empty, $P'$ can consume its next input symbol $a$ and place $h(a)$ in the buffer.

   (b) If $\delta(q, b, X)$ contains $(p, \gamma)$, where $b$ is in $T$ or $b = \epsilon$, then

   $$\delta'\big((q, bx), \epsilon, X\big)$$

   contains $\big((p, x), \gamma\big)$. That is, $P'$ always has the option of simulating a move of $P$, using the front of its buffer. If $b$ is a symbol in $T$, then the buffer must not be empty, but if $b = \epsilon$, then the buffer can be empty.

3. Note that, as defined in (7.1), the start state of $P'$ is $(q_0, \epsilon)$; i.e., $P'$ starts in the start state of $P$ with an empty buffer.

4. Likewise, the accepting states of $P'$, as per (7.1), are those states $(q, \epsilon)$ such that $q$ is an accepting state of $P$.

The following statement characterizes the relationship between $P'$ and $P$:

- $(q_0, h(w), Z_0) \overset{*}{\underset{P}{\vdash}} (p, \epsilon, \gamma)$ if and only if $((q_0, \epsilon), w, Z_0) \overset{*}{\underset{P'}{\vdash}} ((p, \epsilon), \epsilon, \gamma)$.

# MODULE-IV

## Turing machine:

Informal Definition:

We consider here a basic model of TM which is deterministic and have one-tape. There are many variations, all are equally powerfull.

The basic model of TM has a finite set of states, a semi-infinite tape that has a leftmost cell but is infinite to the right and a tape head that can move left and right over the tape, reading and writing symbols.

For any input w with |w|=n, initially it is written on the n leftmost (continguous) tape cells. The infinitely many cells to the right of the input all contain a blank symbol, B whcih is a special tape symbol that is not an input symbol. The machine starts in its start state with its head scanning the leftmost symbol of the input w. Depending upon the symbol scanned by the tape head and the current state the machine makes a move which consists of the following:

- writes a new symbol on that tape cell, •
  moves its head one cell either to the left or to the right and
- (possibly) enters a new state.

The action it takes in each step is determined by a transition functions. The machine continues computing (i.e. making moves) until

- it decides to "accept" its input by entering a special state called accept or final state or
- halts without accepting i.e. rejecting the input when there is no move defined.

On some inputs the TM many keep on computing forever without ever accepting or rejecting the input, in which case it is said to "loop" on that input

### Formal Definition :

Formally, a deterministic turing machine (DTM) is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- Q is a finite nonempty set of states.
- $\Gamma$ is a finite non-empty set of tape symbols, callled the tape alphabet of $M$.
- $\Sigma \subseteq \Gamma$ is a finite non-empty set of input symbols, called the input alphabet of $M$.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L \times R\}$ is the transition function of $M$,

- $q_0 \in Q$ is the initial or start state.
- $B \in \Gamma \setminus \Sigma$ is the blank symbol
- $F \subseteq Q$ is the set of final state.

So, given the current state and tape symbol being read, the transition function describes the next state, symbol to be written on the tape, and the direction in which to move the tape head ( $L$ and $R$ denote left and right, respectively ).

## Transition function : $\delta$

- The heart of the TM is the transition function, $\delta$ because it tells us how the machine gets one step to the next.
- when the machine is in a certain state q$\in$Q and the head is currently scanning the tape symbol $X \in \Gamma$ , and if $\delta(q, x) = (p, Y, D)$ , then the machine

1. replaces the symbol $X$ by $Y$ on the tape
2. goes to state $p$, and
3. the tape head moves one cell ( i.e. one tape symbol ) to the left ( or right ) if $D$ is $L$ ( or $R$ ).

The ID (instantaneous description) of a TM capture what is going out at any moment i.e. it contains all the information to exactly capture the "current state of the computations".

It contains the following:

- The current state, $q$
- The position of the tape head,
- The constants of the tape up to the rightmost nonblank symbol or the symbol to the left of the head, whichever is rightmost.

Note that, although there is no limit on how far right the head may move and write nonblank symbols on the tape, at any finite

time, the TM has visited only a finite prefix of the infinite tape.

An ID (or configuration) of a TM $M$ is denoted by $\alpha q \beta$ where $\alpha, \beta \in \Gamma^*$ and

- $\alpha$ is the tape contents to the left of the head
- $q$ is the current state.
- $\beta$ is the tape contents at or to the right of the tape head

That is, the tape head is currently scanning the leftmost tape symbol of $\beta$. ( Note that if $\beta = \in$ , then the tape head is scanning a blank symbol)

If $q_0$ is the start state and w is the input to a TM M then the starting or initial configuration of M is onviously denoted by $q_0 w$

## Moves of Turing Machines

To indicate one move we use the symbol $\vdash$. Similarly, zero, one, or more moves will be represented by $\vdash$. A move of a TM

$M$ is defined as follows.

Let $\alpha Z q X \beta$ be an ID of $M$ where $X, Z \in \Gamma$, $\alpha, \beta \in \Gamma^*$ and $q \in Q$.

Let there exists a transition $\delta(q, X) = (p, Y, L)$ of $M$.

Then we write $\alpha Z q X \beta \vdash_M \alpha q Z Y \beta$ meaning that ID $\alpha Z q X \beta$ yields $\alpha Z q Y \beta$

- Alternatively , if $\delta(q, X) = (p, Y, R)$ is a transition of $M$, then we write $\alpha Z q X \beta \vdash \alpha z Y p \beta$ which means that the ID $\alpha z q X \beta$ yields $\alpha z Y p \beta$

- In other words, when two IDs are related by the relation $\vdash$, we say that the first one yields the second ( or the second is the result of the first) by one move.
- If IDj results from IDi by zero, one or more (finite) moves then we write $\vdash$ ( If the TM $M$ is understand, then the subscript $M$ can be dropped from $\vdash$ or $\vdash$)

## Special Boundary Cases

- Let $q x \alpha$ be an ID and $\delta(q, x) = (p, Y, L)$ be an transition of $M$. Then $\vdash$. That is, the head is not allowed to fall off the left end of the tape.
- Let $q x \alpha$ be an ID and $\delta(q, x) = (p, Y, R)$ then figure (Note that $\alpha Y q$ is equivalent to $\alpha Y q B$ )
- Let $q x \alpha$ be an ID and $\delta(q, x) = (p, B, R)$ then figure
- Let $\alpha z q x$ be an ID and $\delta(q, x) = (p, B, L)$ then figure

The language accepted by a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, denoted as $L(M)$ is

$L(M)$ = { $w$ | $w \in \Sigma^*$ and figure for some $p \in F$ and $\alpha, \beta \in \Gamma^*$ }

In other words the TM $M$ accepts a string $w \in \Sigma^*$ that cause M to enter a final or accepting state when started in its initial ID (i.e. $q_0 w$). That is a TM $M$ accepts the string $w \in \Sigma^*$ if a sequence of IDs, $ID_1, ID_2, \cdots, ID_k$ exists such that

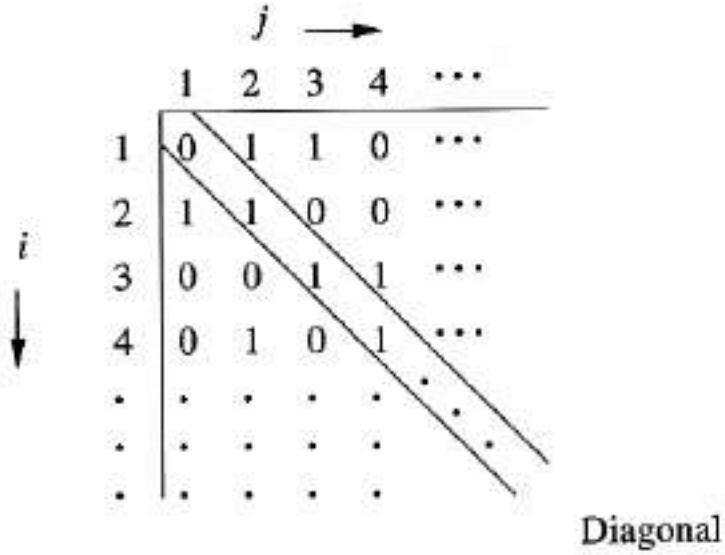
- $ID_1$ is the initial or starting ID of $M$
- $ID_i \vdash_M ID_{i+1}$; $1 \le i < k$

**Diagonalization language:**

- The language $L_d$, the *diagonalization language*, is the set of strings $w_i$ such that $w_i$ is not in $L(M_i)$.

That is, $L_d$ consists of all strings $w$ such that the TM $M$ whose code is $w$ does not accept when given $w$ as input.

The reason $L_d$ is called a "diagonalization" language can be seen if we consider Fig. 9.1. This table tells for all $i$ and $j$, whether the TM $M_i$ accepts input string $w_j$; 1 means "yes it does" and 0 means "no it doesn't."[1] We may think of the $i$th row as the *characteristic vector* for the language $L(M_i)$; that is, the 1's in this row indicate the strings that are members of this language.




This table represents language acceptable by Turing machine

The diagonal values tell whether $M_i$ accepts $w_i$. To construct $L_d$, we complement the diagonal. For instance, if Fig. 9.1 were the correct table, then the complemented diagonal would begin $1, 0, 0, 0, \ldots$. Thus, $L_d$ would contain $w_1 = \epsilon$, not contain $w_2$ through $w_4$, which are 0, 1, and 00, and so on.

The trick of complementing the diagonal to construct the characteristic vector of a language that cannot be the language that appears in any row, is called *diagonalization*. It works because the complement of the diagonal is

**Post's Correspondence Problem (PCP)**

A post correspondence system consists of a finite set of ordered pairs $(x_i, y_i),\ i = 1, 2, \cdots, n,$ where $x_i, y_i \in \Sigma^+$ for some alphabet $\Sigma$.

Any sequence of numbers $i_1, i_2, \cdots, i_k \quad s-t.$

is called a solution to a Post Correspondence System.

$x_{i_1}, x_{i_2}, \cdots, x_{i_k} = y_{i_1}, y_{i_2}, \cdots, y_{i_k}$ The Post's Correspondence Problem is the problem of determining whether a Post Correspondence system has a solutions.

Example 1 : Consider the post correspondence system

$\{(aa, aab), (bb, ba), (abb, b)\}$ The list 1,2,1,3 is a solution to it.

Because

$x_1 x_2 x_1 x_3 = y_1 y_2 y_1 y_3$

$\underbrace{aa}_{x_1}\underbrace{bb}_{x_2}\underbrace{aa}_{x_1}\underbrace{abb}_{x_3} = \underbrace{aab}_{y_1}\underbrace{ba}_{y_2}\underbrace{aab}_{y_1}\underbrace{b}_{y_3}$

$aabbaaabb = aabbaaabb$

| $i$ | $x_i$ | $y_i$ |
|---|---|---|
| 1 | $aa$ | $aab$ |
| 2 | $bb$ | $ba$ |
| 3 | $abb$ | $b$ |

(A post correspondence system is also denoted as an instance of the PCP)

Example 2 : The following PCP instance has no solution

| $i$ | $x_i$ | $y_i$ |
|---|---|---|
| 1 | $aab$ | $aa$ |
| 2 | $a$ | $baa$ |

This can be proved as follows. $(x_2, y_2)$ cannot be chosen at the start, since than the LHS and RHS would differ in the first symbol ( $a$ in LHS and $'b'$ in RHS). So, we must start with $(x_1, y_1)$. The next pair must be $(x_2, y_2)$ so that the 3 rd symbol in the RHS becomes identical to that of the LHS, which is a . After this step, LHS and RHS are not matching. If $(x_1, y_1)$ is selected next, then would be mismatched in the 7 th symbol

( $b$ in LHS and $a$ in RHS). If $(x_2, y_2)$ is selected, instead, there will not be any choice to match the both side in the next step.

Example3 : The list 1,3,2,3 is a solution to the following PCP instance.

| $i$ | $x_i$ | $y_i$ |
|---|---|---|
| 1 | 1 | 101 |
| 2 | 10 | 00 |
| 3 | 011 | 11 |

The following properties can easily be proved.

Proposition The Post Correspondence System

$\left\{\left(a^{i_1}, a^{j_1}\right), \left(a^{i_2}, a^{j_2}\right), \cdots, \left(a^{i_n}, a^{j_n}\right)\right\}$ has solutions if and only if

$\exists k$ such that $i_k = j_k$ or
$\exists k$ and $l$ such that $i_k > j_k$ and $i_l < j_l$

Corollary : PCP over one-letter alphabet is decidable.

Proposition Any PCP instance over an alphabet $\Sigma$ with $|\Sigma| \geq 2$ is equivalent to a PCP instance over an alphabet $\Gamma$ with $|\Gamma| = 2$

Proof : Let $\Sigma = \{a_1, a_2, \cdots, a_k\}, k > 2.$

Consider $\Gamma = \{0,1\}$ We can now encode every $a_i \in \Sigma, 1 \leq i \leq k$ as $10^i 1.$ any PCP instance over $\Sigma$ will now have only two symbols, 0 and 1 and, hence, is equivalent to a PCP instance over $\Gamma$

Theorem : PCP is undecidable. That is, there is no algorithm that determines whether an arbitrary Post Correspondence System has a solution.

Proof: The halting problem of turning machine can be reduced to PCP to show the undecidability of PCP. Since halting problem of TM is undecidable (already proved), This reduction shows that PCP is also undecidable. The proof is little bit lengthy and left as an exercise.

Some undecidable problem in context-free languages

We can use the undecidability of PCP to show that many problem concerning the context-free languages are undecidable. To prove this we reduce the PCP to each of these problem. The following discussion makes it clear how PCP can be used to serve this purpose.

Let $\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$ be a Post Correspondence System over the alphabet $\Sigma$. We construct two CFG's $G_x$ and $G_y$ from the ordered pairs *x,y* respectively as follows.

$$G_x=(N_x,\Sigma_x,P_x,S_x)$$ and

$$G_y=(N_y,\Sigma_y,P_y,S_y)$$ where

$$G_y=(N_y,\Sigma_y,P_y,S_y)$$

$$N_x=\{S_x\} \text{ and } N_y=\{S_y\}$$

$$\Sigma_x=\Sigma_y=\Sigma\cup\{1,2,\cdots,n\},$$

$$P_x=\{S_x\to x_iS_xi,\ S_x\to x_ii \mid i=1,2,\cdots,n\}$$

and $P_y=\{S_y\to y_iS_yi,\ S_y\to y_ii \mid i=1,2,\cdots n\}$

it is clear that the grammar $G_x$ generates the strings that can appear in the LHS of a sequence while solving the PCP followed by a sequence of numbers. The sequence of number at the end records the sequence of strings from the PCP instance (in reverse order) that generates the string. Similarly, $G_y$ generates the strings that can be obtained from the RHS of a sequence and the corresponding sequence of numbers (in reverse order).

Now, if the Post Correspondence System has a solution, then there must be a sequence

$$i_1i_2,\cdots,i_k \ s.t$$

$$x_{i_1}x_{i_2}\cdots x_{i_k}=y_{i_1}y_{i_2}\cdots y_{i_k}$$

According to the construction of $G_x$ and $G_y$

$$S_x\overset{*}{\underset{G_x}{\Rightarrow}}x_{i_1}x_{i_2}\cdots x_{i_k}i_ki_{k-1}\ldots i_2i_1 \text{ and}$$

$$S_y\overset{*}{\underset{G_y}{\Rightarrow}}y_{i_1}y_{i_2}\cdots y_{i_k}i_ki_{k-1}\cdots i_2i_1$$

In this case

iii. Is $L(G_1) \subseteq L(G_2)$?

Proof :

i. If $L(G_1) = \Sigma^*$ then, $\overline{L(G_1)} = \phi$

Hence, it suffice to show that the question “Is $L(G_1) = \phi$?" is undecidable.

Since, $\overline{L(G_x)}$ and $\overline{L(G_y)}$ are CFl's and CFL's are closed under union, $L = \overline{L(G_x)} \cup \overline{L(G_y)}$ is also context-free. By DeMorgan's theorem, $\overline{L} = L(G_x) \cap L(G_y)$

If there is an algorithm to decide whether $L(G_1) = \phi$ we can use it to decide whether $\overline{L} = L(G_x) \cap L(G_y) = \phi$ or not. But this problem has already been proved to be undecidable.

Hence there is no such algorithm to decide or not. $L(G_1) = \phi$

ii.

Let *P* be any arbitrary Post correspondence system and $G_x$ and $G_y$ are CFg's constructed from the pairs of strings.

$L_1 = \overline{L(G_x)} \cup \overline{L(G_y)}$ must be a CFL and let $G_1$generates $L_1$. That is,

$$L_1 = L(G_1) = \overline{L(G_x)} \cup \overline{L(G_y)} = \overline{L(G_x) \cap L(G_y)}$$

by De Morgan's theorem, as shown already, any string, $w \in L(G_x) \cap L(G_y)$ represents a solution to the PCP. Hence, $L(G_1)$ contains all but those strings representing the solution to the PCP.

Let $L(G_2) = (\Sigma \cup \{1, 2, \cdots, n\})^*$ for same CFG $G_2$.

It is now obvious that $L(G_1) = L(G_2)$ if and only if the PCP has no solutions, which is already proved to be undecidable. Hence, the question “Is $L(G_1) = L(G_2)$?" is undecidable.

iii.

**Class p-problem solvable in polynomial time:**

A Turing machine $M$ is said to be of *time complexity* $T(n)$ [or to have "running time $T(n)$"] if whenever $M$ is given an input $w$ of length $n$, $M$ halts after making at most $T(n)$ moves, regardless of whether or not $M$ accepts. This definition applies to any function $T(n)$, such as $T(n) = 50n^2$ or $T(n) = 3^n + 5n^4$; we shall be interested predominantly in the case where $T(n)$ is a polynomial in $n$. We say a language $L$ is in class $\mathcal{P}$ if there is some polynomial $T(n)$ such that $L = L(M)$ for some deterministic TM $M$ of time complexity $T(n)$.

**Non deterministic polynomial time:**

A nondeterministic TM that never makes more than p(n) moves in any sequence of choices for some polynomial p is said to be non polynomial time NTM.

- NP is the set of languags that are accepted by polynomial time NTM's
- Many problems are in NP but appear not to be in p.
- One of the great mathematical questions of our age: is there anything in NP that is not in p?

**NP-complete problems:**

If We cannot resolve the "p=np question, we can at least demonstrate that certain problems in NP are the hardest , in the sense that if any one of them were in P , then P=NP.

- These are called NP-complete.
- Intellectual leverage: Each NP-complete problem's apparent difficulty reinforces the belief that they are all hard.

**Methods for proving NP-Complete problems:**

- Polynomial time reduction (PTR): Take time that is some polynomial in the input size to convert instances of one problem to instances of another.
- If P1 PTR to P2 and P2 is in P1 the so is P1.
- Start by showing every problem in NP has a PTR to Satisfiability of Boolean formula.
- Then, more problems can be proven NP complete by showing that SAT PTRs to them directly or indirectly.