Introduction

Purpose of the Theory of Computation: Develop formal mathematical models of computation that reflect real-world computers. Nowadays, the Theory of Computation can be divided into the following three areas:

• Automata Theory

• Computability Theory

• Complexity Theory,

**Automata theory**

Automata Theory deals with definitions and properties of different types of “computation models”. Examples of such models are:

• Finite Automata. These are used in text processing, compilers, and hardware design.

• Context-Free Grammars. These are used to define programming languages and in Artificial Intelligence.

• Turing Machines. These form a simple abstract model of a “real” computer, such as your PC at home.

Central Question in Automata Theory: Do these models have the same power, or can one model solve more problems than the other?

**Computability theory**

In the 1930’s, G¨odel, Turing, and Church discovered that some of the fundamental mathematical problems cannot be solved by a “computer”. (This may sound strange, because computers were invented only in the 1940’s).

An example of such a problem is “Is an arbitrary mathematical statement true or false?” To attack such a problem, we need formal definitions of the notions of

• computer,

• algorithm, and

• computation.

The theoretical models that were proposed in order to understand solvable and unsolvable problems led to the development of real computers.

Central Question in Computability Theory: Classify problems as being solvable or unsolvable.

**Complexity Theory**

The main question asked in this area is “What makes some problems computationally hard and other problems easy?” Informally, a problem is called “easy”, if it is efficiently solvable. Examples of “easy” problems are (i) sorting a sequence of, say, 1,000,000 numbers, (ii) searching for a name in a telephone directory, and (iii) computing the fastest way to drive from Ottawa to Miami. On the other hand, a problem is called “hard”, if it cannot be solved efficiently, or if we don’t know whether it can be solved efficiently. Examples of “hard” problems are (i) time table scheduling for all courses at Carleton, (ii) factoring a 300-digit integer into its prime factors, and (iii) computing a layout for chips in VLSI.

Central Question in Complexity Theory: Classify problems according to their degree of “difficulty”. Give a rigorous proof that problems that seem to be “hard” are really “hard”.

This course

•This course is about the fundamental capabilities and limitations of computers. These topics form the core of computer science.

• It is about mathematical properties of computer hardware and software.

• This theory is very much relevant to practice, for example, in the design of new programming languages, compilers, string searching, pattern matching, computer security, artificial intelligence, etc., etc.

• This course helps you to learn problem solving skills. Theory teaches you how to think, prove, argue, solve problems, express, and abstract.

• This theory simplifies the complex computers to an abstract and simple mathematical model, and helps you to understand them better.

• This course is about rigorously analyzing capabilities and limitation of systems.

Turing machine is equivalent in computing power to the digital computer as we know it today and also to all the most general mathematical notions of computation

***Please refer another notes of unit 7 for more details till regular expression.***

**Finite Automata:** It is used to recognize patterns of specific type input. It is the most restricted type of automata which can accept only regular languages (languages which can be expressed by regular expression using OR (+), Concatenation (.), Kleene Closure(\*) like a\*b\*, (a+b) etc.)

**Deterministic FA and Non-Deterministic FA**: In deterministic FA, there is only one move from every state on every input symbol but in Non-Deterministic FA, there can be zero or more than one move from one state for an input symbol.

**Note:**

Language accepted by NDFA and DFA are same.

Power of NDFA and DFA is same.

No. of states in NDFA is less than or equal to no. of states in equivalent DFA.

For NFA with n-states, in worst case, the maximum states possible in DFA is 2n

Every NFA can be converted to corresponding DFA.

Identities of Regular Expression :

Φ + R = R + Φ = R

Φ \* R = R \* Φ = Φ

ε \* R = R \* ε = R

ε\* = ε

Φ\* = ε

ε + RR\* = R\*R + ε = R\*

(a+b)\* = (a\* + b\*)\* = (a\* b\*)\* = (a\* + b)\* = (a + b\*)\* = a\*(ba\*)\* = b\*(ab\*)\*

**Moore Machine**: Moore machines are finite state machines with output value and its output depends only on present state.

**Mealy Machine:** Mealy machines are also finite state machines with output value and its output depends on present state and current input symbol.

**Regular Expressions**

Regular Expressions are used to denote regular languages. An expression is regular if:

ɸ is a regular expression for regular language ɸ.

ɛ is a regular expression for regular language {ɛ}.

If a ∈ Σ (Σ represents the input alphabet), a is regular expression with language {a}.

If a and b are regular expression, a + b is also a regular expression with language {a,b}.

If a and b are regular expression, ab (concatenation of a and b) is also regular.

If a is regular expression, a\* (0 or more times a) is also regular.

Regular Expression Regular Languages

set of vovels ( a ∪ e ∪ i ∪ o ∪ u ) {a, e, i, o, u}

a followed by 0 or more b (a.b\*) {a, ab, abb, abbb, abbbb,….}

any no. of vowels followed by any no. of consonants v\*.c\* ( where v – vowels and c – consonants) { ε , a ,aou, aiou, b, abcd…..} where ε represent empty string (in case 0 vowels and o consonants )

Regular Grammar : A grammar is regular if it has rules of form A -> a or A -> aB or A -> ɛ where ɛ is a special symbol called NULL.

Regular Languages : A language is regular if it can be expressed in terms of regular expression.

Closure Properties of Regular Languages

Union : If L1 and If L2 are two regular languages, their union L1 ∪ L2 will also be regular. For example, L1 = {an | n ≥ 0} and L2 = {bn | n ≥ 0}

L3 = L1 ∪ L2 = {an ∪ bn | n ≥ 0} is also regular.

Intersection : If L1 and If L2 are two regular languages, their intersection L1 ∩ L2 will also be regular. For example,

L1= {am bn | n ≥ 0 and m ≥ 0} and L2= {am bn ∪ bn am | n ≥ 0 and m ≥ 0}

L3 = L1 ∩ L2 = {am bn | n ≥ 0 and m ≥ 0} is also regular.

Concatenation : If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular. For example,

L1 = {an | n ≥ 0} and L2 = {bn | n ≥ 0}

L3 = L1.L2 = {am . bn | m ≥ 0 and n ≥ 0} is also regular.

Kleene Closure : If L1 is a regular language, its Kleene closure L1\* will also be regular. For example,

L1 = (a ∪ b)

L1\* = (a ∪ b)\*

Complement : If L(G) is regular language, its complement L’(G) will also be regular. Complement of a language can be found by subtracting strings which are in L(G) from all possible strings. For example,

L(G) = {an | n > 3}

L’(G) = {an | n <= 3}

Note : Two regular expressions are equivalent if languages generated by them are same. For example, (a+b\*)\* and (a+b)\* generate same language. Every string which is generated by (a+b\*)\* is also generated by (a+b)\* and vice versa.

How to solve problems on regular expression and regular languages?

Question 1 : Which one of the following languages over the alphabet {0,1} is described by the regular expression?

(0+1)\*0(0+1)\*0(0+1)\*

(A) The set of all strings containing the substring 00.

(B) The set of all strings containing at most two 0’s.

(C) The set of all strings containing at least two 0’s.

(D) The set of all strings that begin and end with either 0 or 1.

Solution : Option A says that it must have substring 00. But 10101 is also a part of language but it does not contain 00 as substring. So it is not correct option.

Option B says that it can have maximum two 0’s but 00000 is also a part of language. So it is not correct option.

Option C says that it must contain atleast two 0. In regular expression, two 0 are present. So this is correct option.

Option D says that it contains all strings that begin and end with either 0 or 1. But it can generate strings which start with 0 and end with 1 or vice versa as well. So it is not correct.

Question 2 : Which of the following languages is generated by given grammar?

S -> aS | bS | ∊

(A) {an bm | n,m ≥ 0}

(B) {w ∈ {a,b}\* | w has equal number of a’s and b’s}

(C) {an | n ≥ 0} ∪ {bn | n ≥ 0} ∪ {an bn | n ≥ 0}

(D) {a,b}\*

Solution : Option (A) says that it will have 0 or more a followed by 0 or more b. But S -> bS => baS => ba is also a part of language. So (A) is not correct.

Option (B) says that it will have equal no. of a’s and b’s. But But S -> bS => b is also a part of language. So (B) is not correct.

Option (C) says either it will have 0 or more a’s or 0 or more b’s or a’s followed by b’s. But as shown in option (A), ba is also part of language. So (C) is not correct.

Option (D) says it can have any number of a’s and any numbers of b’s in any order. So (D) is correct.

Question 3 : The regular expression 0\*(10\*)\* denotes the same set as

(A) (1\*0)\*1\*

(B) 0 + (0 + 10)\*

(C) (0 + 1)\* 10(0 + 1)\*

(D) none of these

Solution : Two regular expressions are equivalent if languages generated by them are same.

Option (A) can generate all strings generated by 0\*(10\*)\*. So they are equivalent.

Option (B) string null can not generated by given languages but 0\*(10\*)\* can. So they are not equivalent.

Option (C) will have 10 as substring but 0\*(10\*)\* may or may not. So they are not equivalent.

Question 4 : The regular expression for the language having input alphabets a and b, in which two a’s do not come together:

(A) (b + ab)\* + (b +ab)\*a

(B) a(b + ba)\* + (b + ba)\*

(C) both options (A) and (B)

(D) none of the above

Solution:

Option (C) stating both both options (A) and (B) is the correct regular expression for the stated question.

The language in the question can be expressed as L={&epsilon,a,b,bb,ab,aba,ba,bab,baba,abab,…}.

In option (A) ‘ab’ is considered the building block for finding out the required regular expression.(b + ab)\* covers all cases of strings generated ending with ‘b’.(b + ab)\*a covers all cases of strings generated ending with a.

Applying similar logic for option (B) we can see that the regular expression is derived considering ‘ba’ as the building block and it covers all cases of strings starting with a and starting with b.

**Grammar in Automata-**

**Formal Definition-**

A Grammar is a 4-tuple such that-

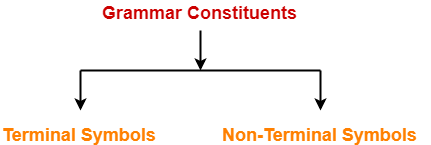
**G = (V , T , P , S)**

where-

* V = Finite non-empty set of non-terminal symbols
* T = Finite set of terminal symbols
* P = Finite non-empty set of production rules
* S = Start symbol

**Grammar Constituents-**

A Grammar is mainly composed of two basic elements-



1. Terminal symbols

2. Non-terminal symbols

**1. Terminal Symbols-**

* Terminal symbols are those which are the constituents of the sentence generated using a grammar.
* Terminal symbols are denoted by using small case letters such as a, b, c etc.

**2. Non-Terminal Symbols-**

* Non-Terminal symbols are those which take part in the generation of the sentence but are not part of it.
* Non-Terminal symbols are also called as **auxiliary symbols** or **variables**.
* Non-Terminal symbols are denoted by using capital letters such as A, B, C etc.

**Examples of Grammar-**

**Example-01:**

Consider a grammar G = (V , T , P , S) where-

* V = { S } // Set of Non-Terminal symbols
* T = { a , b } // Set of Terminal symbols
* P = { S → aSbS , S → bSaS , S → ∈ } // Set of production rules
* S = { S } // Start symbol

This grammar generates the strings having equal number of a’s and b’s

**Example-02:**

Consider a grammar G = (V , T , P , S) where-

* V = { S , A , B } // Set of Non-Terminal symbols
* T = { a , b } // Set of Terminal symbols
* P = { S → ABa , A → BB , B → ab , AA → b } // Set of production rules
* S = { S } // Start symbol

**Types of Grammars-**

Grammars are classified on different basis as-

