

## Lecture 1.1: Introduction to the Theory of Computation

*Presenter: Azwad Anjum Islam (AAI)*

*Scribe: Mujtahid Al-Islam Akon (AKO)*

### Introduction: why this course

We are living in a fascinating world led by a mysterious machine called computers. Our life has become so dependent on these machines that nowadays many people even believe that one day may come when computers will be able to do everything as seen in the science fiction. This might seem a legit guess to many but is that true? Is there any proof that a computer will eventually be invincible solving all the problems that can be fed into it? In short, is there any proven limitation of a computer? If any, then what are they? If these questions have been bugging you, then congratulations, you have come to the right place!

Welcome to this exciting course of computer science where we shall explore, for the first time, the extraordinary science behind computers. In this course, you shall explore the fundamental capabilities of a computer as well as its limitations if there is any.

Before diving into the detail, let's explore few basics. In this lecture, we shall discuss a brief overview of the course, its scope as well as some high-level applications.

### Theory of Computation

This course broadly deals with what is called the **Theory of Computation** – a joint branch of computer science & Mathematics – that can be divided into **three** major components-

- **Automata Theory:** the study of **conceptual machines/abstract machines/automata** that can solve logical problems or computational problems. The singular form of automata is **automaton**.
- **Computability Theory:** deals with the very question whether a problem can actually be solved by a computer (an automaton) or not.
- **Computational Complexity Theory:** deals with the **efficiency** of the solution of a computational problem provided that the problem is solvable by a computer.

This efficiency is measured in terms of-

- **Time Complexity:** the time or the number of steps required to solve the problem in terms of input size.
- **Space (memory) Complexity:** the amount of memory required to solve the problem in terms of input size.

In addition to these, there is the **Formal Language Theory** that is very closely related to the Automata Theory.

However, in this course, we shall focus on the Automata Theory and a little bit on the Computability Theory.

## Rational thinking process and Automata Theory

To understand Automata theory, first, we need to understand how the human mind thinks and take decisions logically. (Recall the example of the Stop sign, finding the number of occurrences of 2's etc. from the main lecture). If you are given an instance (example) of a problem as an **input**, you will, first, **process** the input via proper thinking based on the already-known or given set of instructions or knowledge and then come to a decision in the form of an action or a response, formally known as an **output**.

Automata Theory emerged from this **input-processing-output framework** in the early 20th century when mathematicians and logicians tried to develop the concept of a machine that can replicate this thought process – the way we think.

*Automata are abstract models of machines that can perform logical functions on its own based on some predefined rules given a set of inputs.*

However, **not** all problems are solvable in a computer and that is where the Computability Theory comes in which helps us to decide which problem can be solved from a computational point of view and which one cannot.

A very famous example of the non-solvable problems is **the halting problem**. It is basically the question- 'Can a system be developed that can decide by itself whether a computer program will stop at some point or run indefinitely given the description of an algorithm of that program and its corresponding input?' Alan Turing proved that we cannot develop such a machine.

## (Finite) State Machine

Abstract machines can imitate the logical thought processes of the human mind. The following example of a state machine shows how we can achieve that.

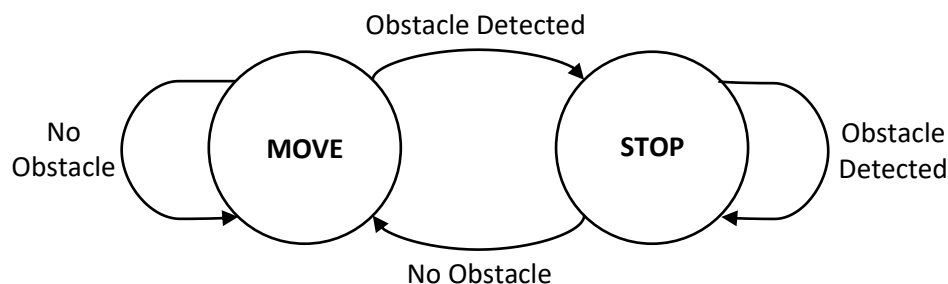


Figure 1: States and corresponding transitions of an obstacle avoiding robot

Consider a simple obstacle avoiding robot. It is built on a very basic principle:

*If there is nothing blocking its path, it keeps **moving**, however, if there is an obstacle, it **stops** right away.*

We can express the above logical statement by a state diagram (refer to CSE260: Digital Logic Design course) consisting of few states and some transitions among them.

Notice, the robot can only be in two possible states (see Figure 1)-

State-1: it is moving (**MOVE** state)

State-2: it is stopped (**STOP** state)

The following way the state diagram simulates the behavior of the robot-

1. Let's start from the MOVE state.
2. As long as it sees no obstacle in its path, it keeps moving i.e. it remains in the same state (the self-loop in MOVE state).
3. The moment the robot's sensor picks up something blocking its path, it goes from MOVE state to the STOP State (following the top arrow from MOVE to STOP).
4. As long as the obstacle is there it remains in that state (the self-loop in STOP state).
5. As soon as the path is cleared, it heads back to the MOVE state again (following the down arrow from STOP to MOVE).

In this way, a state machine can be used to simulate the logical process of an entity.

## Theory of Computation – A look back

- In 1936, Alan Turing first came up with the idea of an abstract machine that had, in theory, all the computational capability of a modern computer.
- It was a simple machine and only had-
  - **A memory tape** of infinite length divided into cells;
  - **A read/write head** that would go back & forth over the tape sequentially and could either write into the associated cell or read from it; and
  - **A control table** that would hold all the predefined rules that would tell the machine what to do in a given situation.
- Turing also tried to define the **boundary** between what a computer potentially could do and what it could not – a subject of discussion in the Computability Theory.

The following points out some important milestones in the history of the theory of computation-

- In 1943, Warren McCulloch and Walter Pitts published a paper that tried to model the nervous system as a finite computational device.
- In the 1950s, G. H. Mealy and E. F. Moore further generalized the theory to more powerful finite machines.
- In the late 1950s, Noam Chomsky started the study of formal grammars. These grammars have close associations with abstract automata and the computability of problems. He also described the famous Chomsky Hierarchy.

- In 1969, S. Cook defined a distinction between the problems that can be solved by computers truly and the problems that can be solved in principle, however, will take so much time – even with sufficient computational power – that computers are essentially useless for those problems.

Interestingly, a lot of the pioneers who contributed highly in this field actually came from other genres-

- The first two people who formalized the finite state machines were neuroscientists.
- Noam Chomsky who pioneered in the field of **formal grammars** – a very important in a number of branches in computer science including Automata theory, Compilers or Natural Language Processing (NLP), and text mining – actually had his background in linguistics and journalism.

## Applications of Theory of Computation

The applicability of Theory of Computation is so immense that S Wolfram, a renowned computer scientist, argues that the whole universe can eventually be described as a machine with a finite number of different states and rules. Some applications follow-

- Finite Automata (FA) and some formal grammars help in designing and construction of important software components and systems.
- Concepts of computability help us decide whether a computational problem can be solved with a direct approach or other workarounds needs to be adopted e.g. simplification, approximation, some kind of heuristics etc.
- Text mining & Pattern matching.
- And many, many more...

In this course, we shall study the concepts of Regular Expression (RE) and Regular Language (RL) which are widely used in pattern matching in many fields. I shall also explore Context-free Grammar (CFG) as well as Context-free Language (CFL) which is the key in recognizing your codes used in various compilers.

---

## Lecture 1.2: Basic concepts and Terminology-1

Presenter: Azwad Anjum Islam (AAI)

Scribe: Mujtahid Al-Islam Akon (AKO)

In this lesson, you will learn about some of the basic concepts and terminologies related to the Automata Theory.

### Families of Automata

There are four major families of automaton based on their capabilities & limitations:

- a. Finite State Machine /Automaton (FSM)
- b. Pushdown Automaton (PDA)
- c. Linear Bounded Automaton (LBA) and
- d. Turing Machine (TM)

These automata are not mutually exclusive. Their proper relationship is depicted below-

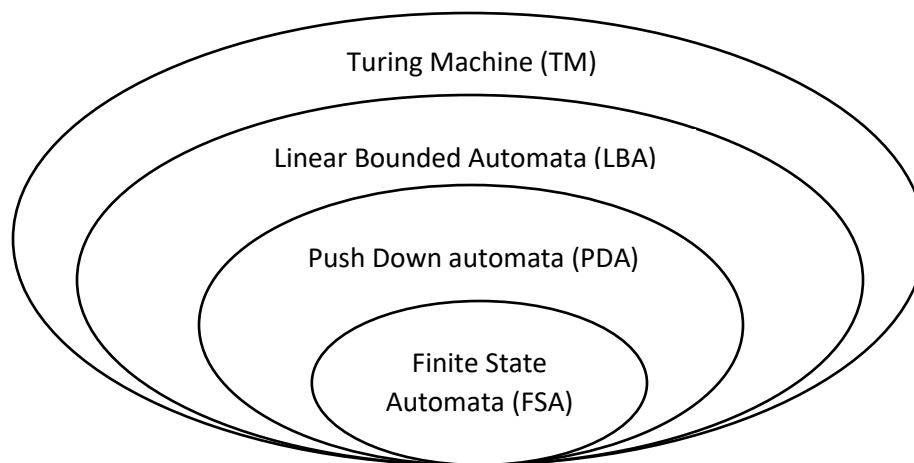


Figure 1: Relation among different families of automata

- The most basic type of automaton is the finite state machine.
- Then the pushdown automaton which encompasses the finite state machine i.e. anything that a finite state machine can do can also be done in a pushdown automaton.
- Then the linear bounded automaton
- At the topmost there is the Turing machine which is the most generic and the most powerful type of automaton. Anything that can be solved using a computer in theory can also be solved using a Turing machine.

The finite state machines can further be divided into two parts-

- a. **Finite state machines with output:** they take an input and do some processing and finally generate an output. These are further divided into-
  - i. Mealy machine

- ii. Moore machine
- b. **Finite state machines without output:** they do not have an output and will only give you either a *yes* or *no* answer. We shall learn this type at length in this course. These can further be divided into two types-
  - i. Deterministic FSA
  - ii. Non-deterministic FSA

## Finite state machines without output

A finite state machine without output works on the following principle-




- First take an **input**
- **Process** the input based on some predefined rules
- Either **accept** (a Yes) or **reject** (a No) the input.

The input can be anything e.g.-







- a string of characters e.g. HELLO
- a stream of numbers e.g. 011011221012
- a string of operations e.g. left right right right. This type of inputs is nothing but a set of strings.
- etc. ...

## Some Definitions

**Symbols:** The smallest building block in the theory of computation and a symbol can be anything e.g.

- Characters like a, b, अ, ९, ∞, £, ÷, 😊
- Numbers like 0, 1, १
- Even pictograms like   
- Etc.

**Strings:** Strings are actually what goes as inputs into the automata. A string is a sequence of symbols e.g.-

- A sequence of characters like hello
- A sequence of numbers like 010010
- A sequence of pictograms      
- Etc.

Few more definitions related to strings:

- **Empty String:** The string that has a length 0 i.e. this string has no characters. This is often denoted by  $\epsilon$ . Remember that an absence of string is still a string which is  $\epsilon$ .
- **Substring:** any consecutive part of a string e.g.  $\epsilon$ , **ALO**, **LOH**, **OMORA**, **ALOHOMORA**, all are some of the substrings generated from the string **ALOHOMORA**.
- **Subsequences:** any part of a string that's in the right order i.e. they are part of the string too, however, they don't have to be consecutive but they must be in the right order.
  - $\epsilon$ , **ALH**, **MR**, **ALORA**, **LOMA**, **ALOHOMORA**, all are subsequences of **ALOHOMORA**.

- **AHL** cannot be a subsequence of **ALOHOMORA** because you cannot find A, H & L in the same order in **ALOHOMORA**.
- **Prefix:** any length of substring starting from the beginning. E.g.  $\epsilon$ , **A**, **AL**, **ALOHO**, **ALOHOMORA** are some of the prefixes of the string **ALOHOMORA**.
- **Suffix:** last part of a string of any length. E.g.  $\epsilon$ , **A**, **RA**, **MORA**, **ALOHOMORA**, each of them is a suffix of the string **ALOHOMORA**.

Notice that  $\epsilon$  and **ALOHOMORA**, each of them is considered a substring, a subsequence, a prefix and a suffix of **ALOHOMORA**, however, they are not **proper substring**, **proper subsequence** etc. They are called **improper subsequences or substrings or prefixes or suffixes**.

- **Concatenation of strings:** Concatenation means joining together. When we concatenate two strings, we put one string after the other. If we concatenate the two strings 'xy' and 'ab' in this order, we will get 'xyab'.
- **Identity of concatenation** is  $\epsilon$  i.e. if we concatenate any string with epsilon on either side of a string, we just end up with string itself.
- **Self-concatenation:** Concatenation can be done with itself.  
Let, a string W be ab (also expressed as  $W^1$ ). Then, concatenation of W with W itself will be abab and can be expressed as WW or  $W^2$ .

Similarly,

$$W^3 = WWW = ababab$$

$$W^4 = WWWW = abababab$$

.  
.
   
.

$$W^\infty = WWW... = ababab...$$

**Find out by yourself: What does  $W^0$  mean?**

**Alphabets:** Alphabet is a finite set of all the symbols that a language can have. Alphabets are often expressed by ' $\Sigma$ '. Some examples of alphabet include-

- Alphabet for English numerals = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Alphabet for English language = {a, b, ..., z, A, B, ... Z, ..., !, ?, ...}

**Language:** a language is a set of string. Any set of strings will be called a language. A language can either be –

- Finite: The number of elements in the language will be finite e.g. {a, bb, aba}
- Infinite: The number of elements will be infinite e.g. {a, aa, aaa, aaaa, ....}, {0, 10, 101, 0100, 0000, ....}, etc.

**All the strings in a language (either finite or infinite) generally maintains a common property.** This property may or may not always be visible but in most cases this common property exists. For example-

- {a, aa, aaa, aaaa, ....} => a string having all possible combinations of 'a'
- {1, 2, 3, 4, 5, ...} => natural numbers
- {0, 1, 10, 11, 100, ...} => binary numbers.

- {6, 28, 496, 8128, ...} => No obvious common property, however, there is one. It is the sequence of all the [perfect numbers](#).

**Languages are generally infinite.** For example- English language has infinite words (strings). However, languages don't always need to be infinite. For example- a baby only knows a certain number of words e.g. two words like papa or mama. So, the language of that child is finite.

---



## Lecture 2.1: Basic concepts and Terminology-2

Presenter: Azwad Anjum Islam (AAI)

Scribe: Mujtahid Al-Islam Akon (AKO)

In this lesson, you will learn some more background concepts and terminologies related to the Automata Theory.

### Languages and their alphabets

- Recall, a language is just a set of strings. For example-
  - $\{\dots, -2, -1, 0, 1, 2, \dots, 5647, 5648, \dots\}$  is a language of all possible integers – from negative infinity to positive infinity.  
**Note:** every member of this set are strings e.g. 0, -2, 5647 etc. might look like single characters or digits or numbers, however, you have to consider them as strings.
  - $\{x, y, xx, xy, yx, yy, \dots\}$  is a language consisting of all possible strings constructed using the symbols x and y.
  - $\{5318008\}$  is a language consisting of only a single member.
  - $\{\epsilon\}$  is also another language with only a member and that is empty string. So, its *cardinality* = 1.
  - $\{\}$  is a language which belongs nothing. This is called an **empty language**. Thus, its *cardinality* = 0.  
**Note:**  $\{\epsilon\}$  and  $\{\}$  might feel similar but they are not. The first one has a member whereas the later does not. Also, compare their cardinality.
- Every language has an alphabet of their own. **Recall**, alphabets are the symbols that might be encountered in any string of that language. if you don't remember what alphabets are alphabets are just set of symbols. For example-
  - For Bangla language, there is an infinite number of strings all of which are built from the alphabet set,  $\Sigma = \{\text{অ, আ, ..., ক, খ, ...}\}$
  - The language of all possible binary strings,  $L = \{0, 1, 00, 01, 10, 11, \dots, 10101110, \dots\}$   
 So, its corresponding alphabet set be  $\Sigma = \{0, 1\}$
  - The language of all valid phone numbers,  $L = \{01710101010, +8801710101010, 01710 - 101010, \dots\}$ . So, its corresponding alphabet set be  $\Sigma = \{0, 1, 2, \dots, 9, -, +\}$
- A language is generally infinite, however, most of the cases, the alphabet sets are finite.

## Relationship between a finite state machine and its language

- The following is a generic finite state machine without output-

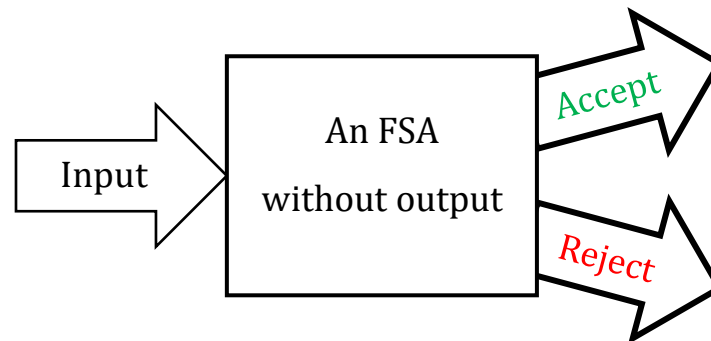


Figure 1: A generic finite state machine without output

- It takes strings as inputs. After processing, some of those inputs gets accepted and some gets rejected. Let's define **an FSA that accepts positive integers of length 2** and find out its inputs, its corresponding language and the alphabet set-
  - Define input:** Let the machine is called **A**. Recall that inputs are valid strings that we decide to feed into machine **A**. For the above problem, all the **integers** can be its probable inputs. So, let's assume that only integers numbers can be fed into this machine. Note that integer set is defined as,  $Z = \{-\infty, \dots, -2, -1, 0, 1, 2, \dots, \infty\}$ . So, 0, 12, 197, 54, 89, 45, -15 etc. can all be the valid inputs for this machine. however, strings like 123.12 (a decimal fraction),  $\forall \partial^{\circ} \text{C} \text{X} \text{E} \text{J}$  or *Dhaka* are not valid input strings for this machine.
  - Define alphabet set:** For machine **A**, only the symbols from 0 to 9 and a (+) sign and a (-) sign can be used to construct the input. So, the alphabet set for this machine is,  $\Sigma = \{1, 2, 3, \dots, 9, +, -\}$ .
  - Define the language:** Among the valid input strings, the machine will **accept** only those string which are positive and whose length is exactly two, and **reject** the others. So, 12, 54, 89, 45 will be accepted, however, 0, 197 and -15 will be rejected. *The complete set of Inputs accepted by the machine is called the language of that finite state machine which is denoted by  $L(A)$ .* So, for machine **A**, the language will be,  $L(A) = \{12, 54, 89, 45, \dots\}$ .

**Find out by yourself:** Is  $L(A)$  for the above machine finite or infinite?

## Different operations on the Language

Languages are sets, so, all the set operations (e.g. union, intersection, complement, etc.) are also applicable on languages.

- The Universal Language/Set:** The language that contains all possible strings that can be generated using the symbols in the alphabets is called the universal language (denoted by  $U(L)$  or  $U$ ). For example-

- For the alphabet,  $\Sigma = \{a, b\}$  of a Language  $L$ , the universal language will be  $U = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$

**Note 1:** The empty string,  $\epsilon$  is the member of the all the universal sets.

**Note 2:** The language,  $L$  is the subset of the universal set,  $U(L)$ .

- **Union of Two Languages:** If a language  $L_1$  has some strings, and another language  $L_2$  has some more strings, then their union language,  $L_{union} = L_1 \cup L_2$ , will have all the strings that are either present in  $L_1$  or in  $L_2$ . For example-

- If  $L_1 = \{a, aa, aaa, \dots\}$  and  $L_2 = \{a, an, the\}$ , then their union,  

$$L_{union} = L_1 \cup L_2 = \{a, an, the, aa, aaa, \dots\}$$

**Note:** In the above example, Alphabet for  $L_1$  is  $\Sigma_1 = \{a\}$  and for  $L_2$  is  $\Sigma_2 = \{a, e, h, n, t\}$ . So, alphabet set of their union set  $L_{union}$  will be,

$$\Sigma_{union} = \Sigma_1 \cup \Sigma_2 = \{a, e, h, n, t\}$$

- **Intersection of Two Languages:** If a language,  $L_1$  has some strings, and another language,  $L_2$  has some more strings, then their intersection language,  $L_{intersection} = L_1 \cap L_2$ , will have only those strings that are present in both  $L_1$  and  $L_2$ .

- If  $L_1 = \{a, aa, aaa, \dots\}$  and  $L_2 = \{a, an, the\}$ , then their intersection,  

$$L_{intersection} = L_1 \cap L_2 = \{a\}$$

**Note:** In the above example, Alphabet for  $L_1$  is  $\Sigma_1 = \{a\}$  and for  $L_2$  is  $\Sigma_2 = \{a, e, h, n, t\}$ . So, alphabet set of their intersection set  $L_{intersection}$  will be,

$$\Sigma_{intersection} = \Sigma_1 \cap \Sigma_2 = \{a\}$$

- **Complement of a Language:** If a language,  $L$  has some strings, its complement language,  $\bar{L}$  will contain all the other strings of the universal set  $U(L)$ , i.e., the strings that are not present in  $L$ .

- If  $L = \{\epsilon, a, aa\}$  and its alphabet  $\Sigma = \{a\}$ , then, the universal language,  

$$U(L) = \{\epsilon, a, aa, aaa, aaaa, aaaaa, \dots\}$$

So, the complement language of  $L$  be-

$$\bar{L} = U - L = \{aaa, aaaa, aaaaa, \dots\}$$

**Note:** As the same set of symbols is used in both  $L$  as well as  $\bar{L}$ , the alphabet for  $L$  and  $\bar{L}$  both will be the same. i.e.

$$\Sigma_{complement} = \Sigma = \{a\}$$

- **Concatenation of Two Languages:** Concatenation means joining together the strings from the two languages. It is similar to the concatenation of two sets that is already discussed (refer to lecture-1.2).

- Let, the language,  $A = \{a, bb\}$  and the language  $B = \{00, 10, 110\}$ , then the concatenation of  $A$  and  $B$  will be,

$$A \cdot B = \{a00, a10, a110, bb00, bb10, bb110\}$$

and the concatenation of  $B$  and  $A$  will be,

$$B \cdot A = \{00a, 00bb, 10a, 10bb, 110a, 110bb\}$$

**Note:** In the above alphabet for  $A$  is  $\Sigma_A = \{a, b\}$  and for  $B$  is  $\Sigma_B = \{0, 1\}$ . So, the alphabet set of the concatenation of  $A$  and  $B$  will be,

$$\Sigma_{A \cdot B} = \Sigma_A \cup \Sigma_B = \{a, b, 0, 1\}$$

- **Self-concatenation of a Language:** Self-concatenation means to concatenate a Language with itself.

- Let,  $D$  be a set, defined as,  $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$  (all single digit numbers). Now, the self-concatenation of  $D$  will be,  $D \cdot D = \{00, 01, 02, 03, \dots, 10, 11, 12, 13, \dots, 99\}$ .  $D \cdot D$  is often written as  $D^2$ .

Therefore,

$$\begin{aligned}
 D^2 &= D \cdot D \\
 &= \{00, 01, 02, 03, \dots, 10, 11, 12, 13, \dots, 99\} \\
 &= \{\text{all 2-digit positive integers}\} \\
 D^3 &= D \cdot D \cdot D \\
 &= D^2 \cdot D \\
 &= \{000, 001, 002, \dots, 100, 101, \dots, 999\} \\
 &= \{\text{all 3-digit positive integers}\} \\
 D^4 &= D \cdot D \cdot D \cdot D \\
 &= D^3 \cdot D \\
 &= \{0000, 0001, 0002, \dots, 1000, \dots, 9999\} \\
 &= \{\text{all 4-digit positive integers}\} \\
 \therefore D^n &= \{\text{all } n\text{-digit positive integers}\}
 \end{aligned}$$

Note that,

$$\begin{aligned}
 D^0 &= \{\text{all 0-digit positive integers}\} \\
 &= \{\epsilon\}
 \end{aligned}$$

- If we take the union of all these sets/languages, we get **the Kleene Closure** (denoted by an asterisk,\*). So, for the above example,

$$D^* = D^0 \cup D^1 \cup D^2 \cup D^3 \cup \dots \cup D^\infty$$

- The Kleene Closure:** Formally, if  $L$  is a set/language, the Kleene Closure of  $L$  is –

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots \cup L^\infty$$

- Example:** If  $L = \{a, bc\}$ , then

$$\begin{aligned}
 L^0 &= \{\epsilon\} \\
 L^1 &= \{a, bc\} \\
 L^2 &= \{aa, abc, bca, bcbc\} \\
 L^3 &= L^1 \cup L^2 \\
 &= \{a, bc\} \cup \{aa, abc, bca, bcbc\} \\
 &= \{aaa, aabc, abca, abcbc, bcaa, bcabc, bcbca, bcbcbc\} \\
 \therefore L^* &= L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots \cup L^\infty \\
 &= \{\epsilon\} \cup \{a, bc\} \cup \{aa, abc, bca, bcbc\} \cup \dots \\
 &= \{\epsilon, a, bc, aa, abc, bca, bcbc, aaa, aabc, abca, abcbc, bcaa, bcabc, \dots\}
 \end{aligned}$$

- The Positive Closure:** It is defined by –

$$L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup \dots \cup L^\infty$$

**Note:** The only difference between  $L^+$  and  $L^*$  is that  $L^*$  has an extra element i.e.  $L^0 = \{\epsilon\}$ .

So, we can write,  $L^+ = L^* - \{\epsilon\}$

- Example:** From the above example, as  $L = \{a, bc\}$ , then

$$L^* = \{\epsilon, a, bc, aa, abc, bca, bcbc, aaa, aabc, abca, abcbc, bcaa, bcabc, \dots\}$$

Therefore,

$$\begin{aligned}
 L^+ &= L^* - \{\epsilon\} \\
 &= \{a, bc, aa, abc, bca, bcbc, aaa, aabc, abca, abcbc, bcaa, bcabc, \dots\}
 \end{aligned}$$

- **Interpretation of  $\Sigma^*$ :** Assume that  $\Sigma$  is an alphabet set for a language  $L$ . Now, we get,  
 $\Sigma^0 = \{\epsilon\}$   
 $\Sigma^1$  = the set of the symbols themselves =  $\Sigma$   
 $\Sigma^2$  = the set of all strings of length 2 that can be created using the alphabet,  $\Sigma$ .  
 $\Sigma^3$  = the set of all strings of length 3 that can be created using the alphabet,  $\Sigma$ .  
 $\vdots$   
 $\Sigma^n$  = the set of all strings of length  $n$  that can be created using the alphabet,  $\Sigma$ .  
 So,  $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots \cup \Sigma^\infty$  is the language that will contain all possible strings of any length that can be generated using  $\Sigma$ , i.e. the universal set of  $L$ .

## Summary

- Refer to figure 1. In a finite state machine,  $M$  –
  - It has an alphabet set =  $\Sigma$
  - The input set for  $M$  = all the strings generated by from the symbols of  $\Sigma$  = the universal set =  $\Sigma^*$
  - The set of accepted strings =  $L(M)$
  - The set of strings that the machine rejects =  $\overline{L(M)}$
- **Example:** Consider the language that contains all valid JAVA identifiers. A java identifier must follow the following rules-
  - (a) Identifiers may contain lower-case and upper-case **letters, digits** from 0 to 9, the dollar sign (\$), and the underscore (\_)
  - (b) Identifiers must be of **length one or more**
  - (c) Identifiers must **not start with a digit**.

Let's find out its alphabet and language.

- **Alphabet set:** According to **rule-(a)**, the alphabet set,  $\Sigma$  consists of-
  - The set of letters,  $L = \{a, b, c, \dots, z, A, B, \dots, Z\}$
  - The set of digits,  $D = \{0, 1, 2, \dots, 9\}$
  - The set of other two symbols,  $S = \{\$, \_ \}$

If we combine all of the above, we shall get the alphabet set. Therefore, the alphabet set,

$$\Sigma = L \cup D \cup S = \{a, b, \dots, 0, 1, \dots, \$, \_ \}$$

- **Language set:** According to **rule-(b)**, if we take the positive closure of  $\Sigma$ , we get the language that contains all possible strings of length  $\geq 1$  i.e.

$$\Sigma^+ = (L \cup D \cup S)^+$$

**Note:** The Kleene Closure contains  $\epsilon$ , which is not a valid identifier. So, we consider the Positive Closure here.

However, according to **rule-(c)**, the strings cannot start with a digit i.e. the symbols from the set,  $D$  defined above. So, we concatenate  $(L \cup S)$  before  $\Sigma^+$  to force it to start with either a letter or a symbol. Now we get,

$$(L \cup S)\Sigma^+ = (L \cup S)(L \cup D \cup S)^+$$

Now notice that  $(L \cup S)$  has a length 1 and  $(L \cup D \cup S)^+$  has minimum length 1 too. So,  $(L \cup S)(L \cup D \cup S)^+$  must have a length of at least two i.e. the above expression will fail to generate single length identifiers like  $x, a, \$$  etc. So, to accommodate those, we need to change the positive closure to the Kleene closure which can be  $\epsilon$  allowing the expression's minimum length to be 1. So, our final language be,

$$L(M) = (L \cup S)\Sigma^* = (L \cup S)(L \cup D \cup S)^*$$

For example, to construct a single length identifier,  $x$  –

- $(L \cup S)$  will produce  $x$
- $(L \cup D \cup S)^*$  will produce  $\epsilon$ .

So, the  $L(M)$  will produce  $x \cdot \epsilon = x$ .

---

## Lecture 2.2: Introduction to Regular Expressions

Presenter: Azwad Anjum Islam (AAI)

Scribe: Mujtahid Al-Islam Akon (AKO)

We know that there are four different types of automata. Each automaton has its own language that it recognizes/accepts:

Automaton	Corresponding language
Finite State Automaton	Regular language (RE)
Pushdown Automaton	Context-free language (CFL)
Linear Bound Automaton	Context sensitive language (CSL)

In this lecture, we shall learn more about regular languages and how to express them algebraically in terms of regular expressions.

### Describing a Regular Language

**Regular language:** It is a simple language with a bunch of strings in it. We shall learn it more formally later.

- A simple regular language can be described intuitively in several ways.
  - You can list all the strings of the language in a tabular set format e.g.  $\{ac, ca, abc, cba, ac, ca, abc, cba, abdbc, cbbdda, cdda, adc, \dots\}$ . If the language is too large or infinite, this description is not good.
  - Also, you can start describing the language in words like we do in set builder method. We can express the above language as-
 

*{Containing strings that start with an **a** or a **c**.  
 If it starts with an **a**, then it ends with a **c**,  
 and if it starts with a **c** then it ends with an **a** ...}*

 You can see that the description can be too wordy sometimes but suitable for many simple cases.
  - You can draw an FSA that accepts the above language. Though this method will work perfectly, the drawing something is a bit tedious too. Is there a better way?
  - Lastly, we have Regular Expression which is easy to derive and often concise to write like a mathematical expression.
- Recall, often the strings in a language has something in common i.e. a common property. There is a pattern in all the strings of a language. For example-
  - $\{1, 10, 100, 1000, \dots\} \Rightarrow$  a set of strings starting with a **1** followed by 0 or more number of **0**'s
  - $\{a, aa, aaa, aaaa, \dots\} \Rightarrow$  a set of strings with any number of **a**
  - $\{1, 2, 3, 4, 5, 6, 7, \dots\} \Rightarrow$  a set of all the natural numbers.

Sometimes the pattern is not easy to find. For example-

- $\{are, republic, cream, brew, require, \dots\} \Rightarrow$  a set of strings having a substring **re**

- $\{\$a, locMin, \_num1, \_1204, \dots\} \Rightarrow$  valid java identifiers.
- Whether you can find it or not, there is a pattern in every regular language. This pattern is known as an **Expression**.
- If a language is regular, then its expression is called a regular expression (**RE**) or **RegEx**.
- Every regular language (**RL**) has one or more regular expressions, i.e. RE of a language is not unique.
- What is an RL actually? Before defining RLs, we need to know more about REs.

## The three basic regular operations

- In a regular expression, three types of operators can be present-

Operator name	Symbol
OR Operator	or +
Concatenation Operator	. or <no symbol>
Kleene closure operator	*
Positive closure operator	+

Don't be confused seeing the + sign in both OR and in the positive closure.

- In the positive closure, the + sign is a superscript i.e. it stays at the top of a symbol.
- In the OR operation, + sign sits between two symbols or two strings
- You have seen concatenation and the closure operators in the previous lecture.
- Some other operators may come as well, however, those are mainly derivatives of these three.
- **The OR operator:** It is a binary operator like what you saw in Boolean algebra.
  - **Position:** It sits between two things e.g.  $R_1 | R_2$  where each of  $R_1$  and  $R_2$  can be any string, a symbol or even another RegEx. You can also write them using + sign e.g.  $R_1 + R_2$ .
  - **Meaning:**  $R_1$  or  $R_2$  means that you may take the one from the left or the one from the right. If you take the one from the left, you get  $R_1$ , and if you take the one from the right, you get  $R_2$ .
  - **Examples:**
    - $a | b = \{a, b\}$
    - $aa | b = \{aa, b\}$
    - $hello | hi = \{hello, hi\}$
    - $(0 | 01) | 001 = 0 | 01 | 001 = \{0, 01, 001\}$
    - Etc.
- **The Concatenation operator:** It is a binary operator but it is **NOT** like the **and** operator that you saw in Boolean algebra.
  - **Position:** It sits between two things e.g.  $R_1 \cdot R_2$  where each of  $R_1$  and  $R_2$  can be any string, a symbol or even another RegEx. You can also write  $R_1$  and  $R_2$  side by side without any sign e.g.  $R_1 R_2$ .
  - **Meaning:**  $R_1$  or  $R_2$  means that you take the  $R_1$  first and then append  $R_2$  to the right.
  - **Examples:**
    - $a \cdot b = \{ab\}$
    - $aa \cdot b = \{aab\}$
    - $hello \cdot hi = \{hellohi\}$



- $(0 \cdot 01) \cdot 001 = 0 \cdot 01 \cdot 001 = \{001001\}$
- Etc.

**Note:** You cannot get more than one expression using only concatenation operators as many as you want.

- **Combination of Concatenation and OR**

- If you combine both, you may need to use the distributive law sometimes. For example-
  - $(a|b) \cdot c = \{ac, bc\}$
  - $(a|aa) \cdot (b|c) = \{ab, aab, ac, aac\}$
  - $hello \cdot hi \cdot (oh|no) \cdot bye = \{hellohiohbye, hellohinoby\}$
- Concatenation has higher precedence than OR. For example-
  - $a \cdot b|c$   
 $= (a \cdot b) | c$   
 $= \{ab, c\} \neq \{ac, bc\}$
  - $hello | hi | oh \cdot no | bye$   
 $= hello|hi|(oh \cdot no)|bye$   
 $= \{hello, hi, ohno, bye\}$

- **The Kleene Closure operator:** It is a unary operator i.e. it only works with one operand.

- **Position:** It sits top on something e.g.  $R^*$  where  $R$  can be any string, a symbol or even another RegEx.
- **Meaning:**  $R^*$  means **zero** or more occurrences of  $R$ .
- **Examples:**
  - $a^* = \{\epsilon, a, aa, aaa, aaaa, \dots\}$
  - $(hello)^* = \{\epsilon, hello, hellohello, hellohellohello, \dots\}$

- **The Positive Closure operator:** It is a unary operator i.e. it only works with one operand.

- **Position:** It sits top on something, e.g.  $R^+$  where  $R$  can be any string, a symbol or even another RegEx.
- **Meaning:**  $R^+$  means **one** or more occurrences of  $R$ .
- **Examples:**
  - $a^+ = \{a, aa, aaa, aaaa, \dots\}$
  - $(hello)^+ = \{hello, hellohello, hellohellohello, \dots\}$
  - Note that,  
 $\epsilon^+ = \{\epsilon, \epsilon\epsilon, \epsilon\epsilon\epsilon, \epsilon\epsilon\epsilon\epsilon, \dots\}$   
 $= \{\epsilon, \epsilon, \epsilon, \epsilon, \dots\}$   
 $= \{\epsilon\}$

- Kleene or positive closure has the **highest precedence** of them all, then the Concatenation operator and lastly the OR operator. For example-

- $(a|b^*) \cdot c = (a|(b^*)) \cdot c = \{ac, c, bc, bbc, bbbc, \dots\}$
- $ab | (xy)^* = \{ab, \epsilon, xy, xyxy, xyxyxy, \dots\}$
- $(01)^* = \{\epsilon, 01, 0101, 010101, \dots\}$
- $(0^*|1^*) = \{\epsilon, 0, 00, 000, \dots, 1, 11, 111, \dots\}$
- $(a|b)^* = (a|b) \cdot (a|b) \cdot (a|b) \cdot (a|b) \cdot \dots$   
 $= \{\epsilon, \text{all the strings made with } a \text{ and } b\}$

Informally, any expression that you can create using OR ( $|$ ), Concatenation ( $\cdot$ ), and the Closure ( $^*$ ,  $^+$ ) operators are called regular expressions.

## Equivalence of regular expressions and regular languages

There are two conditions for an expression to be considered as a regular expression of a certain regular language. This conditions as a whole is known as the equivalence of regular expressions and regular language. They are as follows-

1. Any string that matches with the regular expression has to be a member of the regular language.
2. Every string of the language must match with the regular expression.

Thus, the following cases hold-

- If a string matches with the regular expression  $R$ , but is not a member of the regular language  $L$ , then,  $R$  is not the regular expression of  $L$ .
  - Again, if there is a member of  $L$  that does not match with  $R$ , then  $R$  is not the expression of  $L$ .
-