# ***Why TOC?***

The study of the theory of computation has several purposes, most importantly,

* To familiarize with the foundations and principles of computer science
* To strengthen our ability to carry out formal and rigorous mathematical arguments.

The theory of computation, includes several topics like automata theory, formal languages and grammars, computability, and complexity.

Toc just constitutes the theoretical foundation of computer science.

Computer science is a practical discipline. Those who work in it often have a marked preference for useful and tangible problems over theoretical speculation. Theoretical questions interest them only if they help in finding good solutions.

***Then why study theory?***

Theory provides concepts and principles that help us understand the general nature of the discipline. The field of computer science includes a wide range of special topics, from machine design to programming.

The use of computers in the real world involves a wealth of specific detail that must be learned for a successful application. This makes computer science a very diverse and broad discipline. But in spite of this diversity, there are some common underlying principles.

Study these basic principles, it’s possible for us to construct abstract models of computers and computation. These models embody the *important features that are common to both hardware and software and that are essential to many of the special and complex constructs* we encounter while working with computers. The construction of models is one of the essentials of any scientific discipline, and the usefulness of a discipline is often dependent on the existence of simple, yet powerful, theories and laws.

***Traditional central areas of the theory of computation***

* Automata
* Computability
* Complexity

And these areas are linked by the question, ***WHAT ARE THE FUNDAMENTAL LIMITATIONS AND CAPABILITIES OF A COMPUTER***.

***Complexity:***

Computer problems come in different varieties; some are easy, and some are hard. For example, the sorting problem VS scheduling problem.

*Central question of complexity theory:* What makes some problems computationally hard and others easy?

We have several options when we confront a problem that appears to be computationally hard.

First, by understanding which aspect of the problem is at the root of the difficulty, then we may be able to alter it so that the problem is more easily solvable.

Second, we may be able to settle for less than a perfect solution to the problem. In certain cases finding solutions that only approximate the perfect one is relatively easy.

Third, some problems are hard only in the worst case situation, but easy most of the time. Depending on the application, we should be satisfied with a procedure that occasionally is slow but usually runs quickly.

Finally, we may consider alternative types of computation, such as randomized computation, that can speed up certain tasks.

In complexity theory, the objective is to classify problems as easy ones and hard ones.

***Computability:***

Computability theory classify the problems by those that are solvable and those that are not.

***Automata theory:***

Automata theory deals with the *definitions and properties of mathematical models of computation*. These models play a role in several applied areas of computer science.

One model, called the *finite automaton*, is used in *text processing, compilers, and hardware design.*

Another model, called the *context-free grammar*, is used in *programming languages and artificial intelligence.*

# ***Models that represent features at the core of all computers and their applications:***

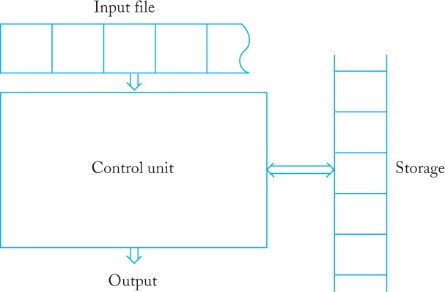
1. Firstly, automata

To model the hardware of a computer, we introduce the notion of an **automaton** (plural, **automata**).

An **automaton** is a construct that possesses all the indispensable features of a digital computer. It accepts input, produces output, may have some temporary storage, and can make decisions in transforming the input into the output.

**Automata**

An automaton is an abstract model of a digital computer. As such, every automaton includes some essential features. It has a mechanism for reading input. It will be assumed that the input is a string over a given alphabet, written on an input file, which the automaton can read but not change.



The input file is divided into cells, each of which can hold one symbol. The input mechanism can read the input file from left to right, one symbol at a time. The input mechanism can also detect the end of the input string (by sensing an end-of-file condition). The automaton can produce output of some form. It may have a temporary **storage** device, consisting of an unlimited number of cells, each capable of holding a single symbol from an alphabet (not necessarily the same one as the input alphabet). The automaton can read and change the contents of the storage cells. Finally, the automaton has a **control unit**, which can be in any one of a finite number of **internal states**, and which can change state in some defined manner.

An automaton is assumed to operate in a discrete timeframe. At any given time, the control unit is in some internal state, and the input mechanism is scanning a particular symbol on the input file. The internal state of the control unit at the next time step is determined by the **next-state** or **transition function**. This transition function gives the next state in terms of the current state, the current input symbol, and the information currently in the temporary storage. During the transition from one time interval to the next, output may be produced or the information in the temporary storage changed. The term **configuration** will be used to refer to a particular state of the control unit, input file, and temporary storage. The transition of the automaton from one configuration to the next will be called a **move**.

A ***deterministic automaton*** is one in which each move is uniquely determined by the current configuration. If we know the internal state, the input, and the contents of the temporary storage, we can predict the future behavior of the automaton exactly.

In a ***nondeterministic automaton***, this is not so. At each point, a nondeterministic automaton may have several possible moves, so we can only predict a set of possible actions.

An automaton whose output response is limited to a simple “yes” or “no” is called an **accepter**. Presented with an input string, an accepter either accepts the string or rejects it. A more general automaton, capable of producing strings of symbols as output, is called a **transducer**.

1. Then, formal language

A **formal language** is an *abstraction of the general characteristics of programming languages*.

A formal language consists of a *set of symbols* and *some rules of formation* by which these symbols can be combined into entities called *sentences*. A formal language is the set of all sentences permitted by the rules of formation.

Formal languages have many of the same essential features of the programming languages and even they are simpler than programming language.

### ***Characteristics of a Good Programming Language?***

There are some popular high-level programming languages, while there are others that could not become so popular in-spite of being very powerful. There might be reasons for the success of a language but one obvious reason is its characteristics. Several characteristics believed to be important for making it good:  
  
A good programming language must be simple and easy to learn and use. It should provide a programmer with a clear, simple and unified set of concepts that can be grasped easily. The overall simplicity of this strongly affects the readability of the programs written in that language and programs that are easier to read and understand are easier to maintain. It is also easy to develop and implement a compiler or an interpreter for a simple language. However, the power needed for the language should not be sacrificed for simplicity. For Example, BASIC is liked by many programmers because of its simplicity.

## ***Naturalness:***

A good language should be natural for the application area for which it is designed. That is, it should provide appropriate operators, data structures, control structures and a natural syntax to facilitate programmers to code their problems easily and efficiently. FORTRAN and COBOL are good examples of languages possessing high degree of naturalness in scientific and business application areas, respectively.

## ***Abstraction:***

Abstraction means ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored. The degree of abstraction allowed by a language directly affects its ease of programming. For Example, object-oriented languages support high degree of abstraction. Hence, writing programs in object-oriented languages is much easier. Object-oriented also support re usability of program segments due to this feature.

1. ***Efficiency:***

Programs written in a good language are translated into machine code efficiently, are executed and require relatively less space in memory. That is, a good programming language is supported with a good language translator (a compiler or an interpreter) that gives due consideration to space and time efficiency.

## ***Structured Programming Support:***

A good language should have necessary features to allow programmers to write their programs based on the concepts of structured programming. This property greatly affects the ease with which a program may be written, tested and maintained. Moreover, it forces a programmer to look at a problem in a logical way so that fewer errors are created while writing a program for the problem.

## ***Compactness:***

In a good language, programmers should be able to express the intended operations concisely without losing readability. Programmers generally do not like a verbose language because they need to write too much. Many programmers dislike COBOL, because it is verbose in nature (Lacks Compactness)

## ***Locality:***

A good language should be such that while writing a program, a programmer need not jump around the visually as the text of a program is prepared. This allows the programmer to concentrate almost solely on the part of the program around the statement currently being worked with. COBOL and to some extent C and Pascal lack locality because data definitions are separated from processing statements, perhaps by many pages of code, or have to appear before any processing statement in the function/procedure.

## ***Extensibility:***

A good language should also allow extensions through a simply, natural and elegant mechanism. Almost all languages provide subprogram definition mechanisms for the purpose, but some languages are weak in this aspect.

## ***Suitability to its Environment****:*

Depending upon the type of application for which a programming language has been designed, the language must also be made suitable to its environment. For Example, a language designed for a real-time applications must be interactive in nature. On the other hand, languages used for data-processing jobs like payroll, stores accounting etc. may be designed to operative in batch mode.

1. Finally, algorithmic computation

Formalizing the concept of a mechanical computation by giving a precise definition of the term **algorithm** and studying the kinds of problems that are (and are not) suitable for solution by such mechanical means.

***BASIC CONCEPTS***

*Languages****:***

What exactly the word “language” means?

A system suitable for the expression of certain ideas, facts, or concepts, including a set of symbols and rules for their manipulation.

*Alphabets and Symbols****:***

We define, an alphabet to be any nonempty finite set.

The members of the alphabet are the symbols of the alphabet. So a finite, nonempty set **Σ** of symbols, called the **alphabet**.

*Strings****:***

From the individual symbols we construct **strings**, which are finite sequences of symbols from the alphabet.

* If Σ1 = {0, 1}, then 01001 is a string over Σ1.
* If Σ2 = {a, b, c, . . . , z}, then abracadabra is a string over Σ2.
* If the alphabet Σ3 = {a, b}, then abab and aaabbba are strings on Σ3.

*Concatenation of two strings****:***

The **concatenation**of two strings w and v is the string obtained by appending the symbols of v to the right end of w, that is, if w = a1a2 ··· an and v = b1b2 ··· bm, then the concatenation of w and v, denoted by wv, is wv = a1a2 ··· anb1b2 ··· bm.

*Reverse of a string****:***

The **reverse** of a string is obtained by writing the symbols in reverse order; if w is a string as shown above, then its reverse **wR** is wR = an ··· a2a1.

*Length of a string****:***

The **length** of a string w, denoted by **|w|**, is the number of symbols in the string.

*Empty string****:***

The string of length zero is called the emptystring and is written **ε**. The empty string plays the role of **0** in a number system.

*Substring****:***

Any string of consecutive symbols in some w is said to be a **substring** of w.

If w=vu, then the substrings v and u are said to be a ***prefix*** and a ***suffix*** of w, respectively. For example, if w = abbab, then {λ, a, ab, abb, abba, abbab} is the set of all prefixes of w, while bab, ab, b are some of its suffixes.

*Length of their concatenation:*

If *u* and *v* are strings, then the *length of their concatenation is the sum of the individual lengths*, that is,|*uv*| = |*u*| + |*v*|.

*Lexicographic ordering of strings:*

The lexicographic ordering of strings is the same as the familiar dictionary ordering, except that shorter strings precede longer strings. Thus the lexicographic ordering of all strings over the alphabet {0, 1} is (ε, O. A, 00, 0, 10, 11, 00, ).

*Repetition of string:*

If w is a string, then wn stands for the string obtained by repeating w n times. As a special case, we define w*0 =*ε*,* for all w.

*Σ\* and Σ+ :*

If Σ is an alphabet, then we use Σ\* to denote the set of strings obtained by concatenating zero or more symbols from Σ. The set Σ\* always contains ε.

To exclude the empty string, we define Σ+ = Σ\* − {ε}.

While Σ is finite by assumption, Σ\* and Σ+ are always infinite since there is no limit on the length of the strings in these sets.

*Notes:*

* A language is defined very generally as a subset of Σ\*.
* A string in a language L will be called a **sentence** of L.
* Any set of strings on an alphabet Σ can be considered a language.

*Example for finite and infinite language:*

* Let Σ = {a, b}. Then Σ\* = {ε, a, b, aa, ab, ba, bb, aaa, aab, ...}.
* The set {a, aa, aab} is a language on Σ. Because it has a finite number of sentences, we call it a finite language.
* The set L = {anbn : n ≥ 0} is also a language on Σ. The strings aabb and aaaabbbb are in the language L, but the string abb is not in L. This language is infinite.

*Language is a Set:*

Since languages are sets, the union, intersection, and difference of two languages are immediately defined.

*Complement of a language:*

The complement of a language is defined with respect to Σ\*; that is, the complement of L is

= Σ\*− L.

*Reverse of a language:*

The reverse of a language is the set of all string reversals, that is, LR = {wR : w ∈ L}.

*Concatenation of a language:*

The concatenation of two languages L1 and L2 is the set of all strings obtained by concatenating any element of L1 with any element of L2; specifically, L1L2 = {xy : x ∈ L1, y ∈ L2}.

We define Ln as L concatenated with itself n times, with the special cases L0 = {λ} and L1 = L for every language L.

*Star closure and positive closure of a language:*

**Star-closure** of a language is defined as L\* = L0 ∪ L1 ∪ L2 ···

And the **positive closure** is defined as L+ = L1 ∪ L2 ···.

*Examples:*

* If L = {anbn : n ≥ 0}, then L2 = {anbnambm : n ≥ 0, m ≥ 0}.
* The reverse of L is easily described in set notation as LR = {bnan : n ≥ 0}

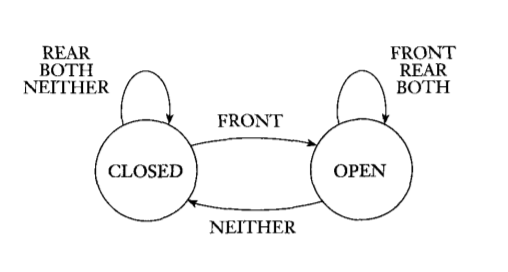
Try, describing and L\*.

FINITE AUTOMATA:

Finite automata or finite state acceptors are good models for computers with an extremely limited amount of memory. It is finite because it has only a finite set of internal states and no other memory. It is called an accepter because it processes strings and either accepts or rejects them

We can think of finite state machine as a *simple pattern recognition mechanism*.

Finite automaton is characterized by having no temporary storage. Since an input file cannot be rewritten, a finite automaton is severely limited in its capacity to “remember” things during the computation. A finite amount of information can be retained in the control unit by placing the unit into a specific state. But since the number of such states is finite, a finite automaton can only deal with situations in which the information to be stored at any time is strictly bounded.

*Example:* The controller for an automatic door is one example of such a device. An automatic door has a pad in front to detect the presence of a person about to walk through the doorway. Another pad is located to the rear of the doorway so that the controller can hold the door open long enough for the person to pass all the way through and also so that the door does not strike someone standing behind it as it opens.

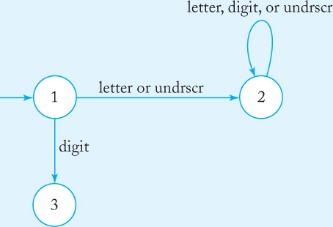
**State diagram for automated controller door**

The rules for variable identifiers in C are

**1.** An identifier is a sequence of letters, digits, and underscores.

**2.** An identifier must start with a letter or an underscore.

**3.** Identifiers allow upper- and lower-case letters.



**An automaton that accepts all legal C identifiers**

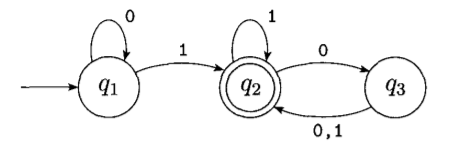
A finite automaton has several parts. It has a set of states and rules for going from one state to another, depending on the input symbol. It has an input alphabet that indicates the allowed input symbols. It has a start state and a set of accept states.

Formal definition:

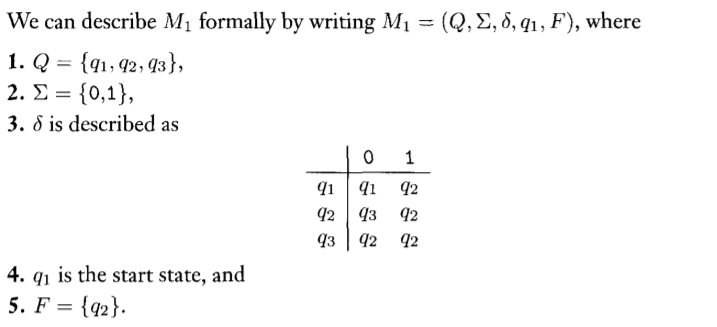
A finite automaton is a 5-tuple (Q , Σ , δ , F),

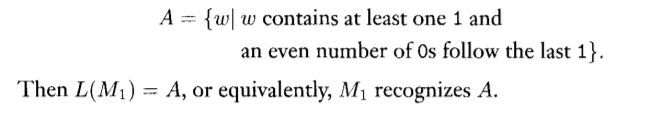
1. Q is a finite set called the states
2. Σ is a finite set called the alphabet
3. δ : Q x Σ Q is the **transition function**
4. Q is the start state
5. F Q is the set of accept states

**EXAMPLE:**



**The finite automaton M1**





If A is the set of all strings that machine M accepts, we say that A is the language of machine M and write L(M) = A. We say that M recognizes A or that M accepts A.

Though a machine may accept several strings, it always recognizes only one language.

Also, even if the machine accepts no strings at all, it still recognizes one language namely, **the empty language** φ.

Formally, if *M* = (*Q*, Σ, *δ, q*0, *F*) is a deterministic finite accepter, then its associated transition graph *GM* has exactly |*Q*| vertices, each one labeled with a different *qi* ∈ *Q*. For every transition rule *δ* (*qi*, *a*) = *qj*, the graph has an edge (*qi*, *qj*) labeled *a*. The vertex associated with *q*0 is called the initialvertex, while those labeled with *qf* ∈ *F* are the finalvertices.

*The extended transition function*:

**δ\* : Q × Σ\* → Q.**

The second argument of δ\* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string.

For example, if *δ* (*q*0, *a*) = *q*1 and *δ* (*q*1, *b*) = *q*2, then *δ*\* (*q*0, *ab*) = *q*2.

That is., δ\* (q, λ) = q

δ\* (q, wa) = δ (δ\* (q, w), a)

δ\* (q0, ab) = δ (δ\* (q0, a), b)

δ\*(q0,a) = δ(δ\*(q0,λ),a) = δ(q0,a) = q1

*δ*\* (*q*0, *ab*) = *δ* (*q*1, *b*) = *q*2

for all q ∈ Q, w ∈ Σ\*, a ∈ Σ.

The language is the set of all the strings accepted by the automaton.

The language accepted by a dfa *M* = (*Q*, Σ, *δ, q*0, *F*) is the set of all strings on Σ accepted by *M*. In formal notation,

*L* (*M*) = {*w* ∈ Σ\* : *δ*\* (*q*0, *w*) ∈ *F*}.