

THEORY OF COMPUTATION

Chapter 1

Introduction to Theory of Computation

1.1 Definition of Languages

1.2 Definition of Grammars

1.3 Definition of Automata

1.4 Some Applications

Practice: 15 MCQs with Answer Key and 10 Broad Questions

Notes in paragraph, list and table format with diagrams, prepared for B.E. / B.Tech / Diploma engineering students (Computer Science, IT and related branches).

Poly Notes Hub

Chapter 1: Introduction to Theory of Computation

Overview of the Subject

The **Theory of Computation (TOC)** is the branch of computer science and mathematics that studies *what can be computed*, *how efficiently it can be computed*, and *which mathematical models describe computation*. Every program you write runs on a real machine, but real machines differ in speed, memory and architecture. TOC strips away these details and studies computation in its purest form using abstract models: **languages, grammars and automata**. Conclusions drawn on these models hold for every real computer, today and in the future.

A useful way to remember the subject is through three questions that every branch of TOC tries to answer:

- **What can be computed at all?** Are there problems that no algorithm can ever solve? (Computability theory)
- **What can be computed efficiently?** Which problems need a reasonable amount of time and memory, and which need an impractical amount? (Complexity theory)
- **What kind of machine is needed for which kind of problem?** How does the power of a machine relate to the kind of language it can recognise? (Automata theory)

Branches of the Theory of Computation

Branch	Central question	Main models / tools
Automata Theory	How do abstract machines work and what languages do they recognise?	Finite automata, pushdown automata, Turing machines
Formal Languages and Grammars	How can sets of strings (languages) be described by finite rules?	Regular expressions, grammars, Chomsky hierarchy
Computability Theory	Which problems are solvable by an algorithm and which are not?	Turing machines, decidability, halting problem
Complexity Theory	How much time and space does a solvable problem need?	Classes P, NP, NP-complete, space classes

Why Engineering Students Study TOC

- It gives the **mathematical foundation** of compiler design, programming languages and operating systems.
- It teaches **formal thinking**: defining a problem precisely, then proving a claim about it.
- It shows the **limits of computing**, for example why no tool can detect every bug in every program.
- It is a **core university subject** and an important topic in competitive examinations such as GATE (CS/IT).

Brief History

Period	Contribution
1930s	Alan Turing (Turing machine) and Alonzo Church (lambda calculus) formalise the idea of an algorithm.
1943 - 1956	McCulloch and Pitts model neurons as simple machines; Kleene introduces regular expressions; Moore and Mealy study finite-state machines.
1956 - 1959	Noam Chomsky proposes the hierarchy of grammars; Rabin and Scott introduce non-deterministic finite automata.
1960s onwards	Context-free grammars are used to define programming languages (ALGOL); complexity theory (P versus NP) develops from the 1970s.

Chapter Map

The four topics of this chapter build on each other. A **language** is the object of study, a **grammar** generates a language, and an **automaton** recognises (accepts) a language. Applications show where these ideas appear in real software and hardware.

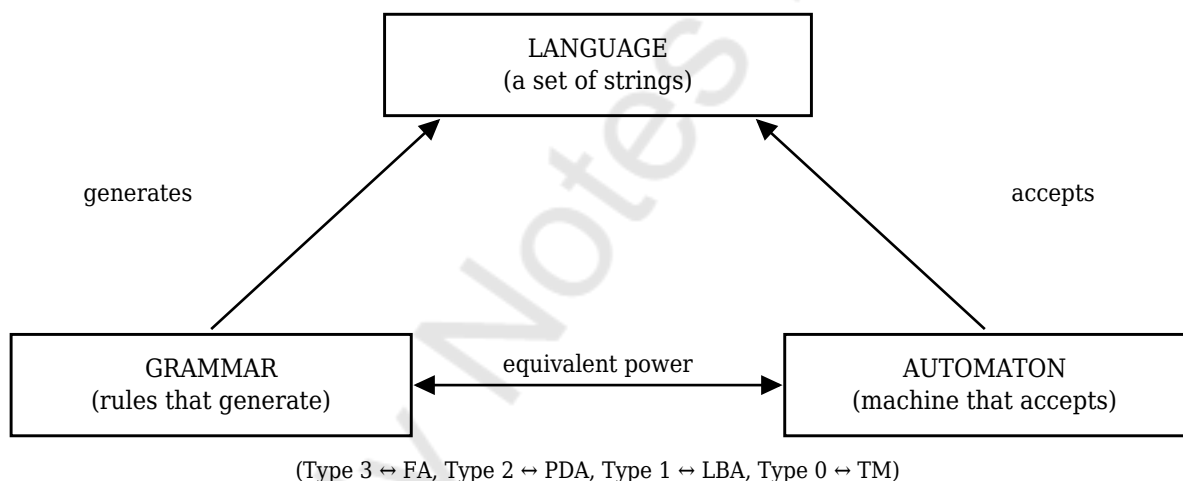


Figure 1.1: Relationship between a language, a grammar and an automaton

1.1 Definition of Languages

1.1.1 Basic Terminology

Before defining a language we need the building blocks from which it is made. These terms are used in every later topic of the subject, so learn them carefully.

Definition: Alphabet

An **alphabet** is a finite, non-empty set of symbols. It is denoted by Σ (sigma).

Definition: String

A **string** (also called a *word*) over Σ is a finite sequence of symbols chosen from Σ . The string with no symbols is the **empty string**, denoted by ϵ (epsilon). The **length** of a string w , written $|w|$, is the number of symbols in it.

Term	Meaning	Example
Symbol	An indivisible character of the alphabet	0, 1, a, b, +
Alphabet Σ	Finite non-empty set of symbols	$\Sigma = \{0, 1\}$, $\Sigma = \{a, b, c\}$, $\Sigma = \{A, \dots, Z\}$
String w	Finite sequence of symbols from Σ	$w = 0110$ over $\{0, 1\}$
Length $ w $	Number of symbols in w	$ 0110 = 4$, $ \epsilon = 0$
Empty string ϵ	String of length zero	$ \epsilon = 0$, and $\epsilon w = w\epsilon = w$
Σ^k	Set of all strings of length exactly k over Σ	For $\Sigma = \{a, b\}$: $\Sigma^2 = \{aa, ab, ba, bb\}$
Σ^*	Set of all strings over Σ , including ϵ (Kleene closure of Σ)	$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$
Σ^+	Set of all non-empty strings over Σ	$\Sigma^+ = \Sigma^* - \{\epsilon\}$

Important points about alphabets

- An alphabet must be **finite**. The set of all natural numbers is infinite, so it cannot be an alphabet.
- $\Sigma^0 = \{\epsilon\}$ for every alphabet Σ , because there is exactly one string of length zero.
- If Σ has k symbols, then the number of strings of length n is k^n . For $\Sigma = \{a, b\}$ there are $2^3 = 8$ strings of length 3.
- Σ^* is always **infinite** (as soon as Σ is non-empty) but **countable**: its strings can be listed one by one in order of length.

1.1.2 Operations on Strings

Operation	Definition	Example ($x = ab, y = ba$)
Concatenation	xy is x followed by y . $ xy = x + y $. Not commutative; associative.	$xy = abba, yx = baab$
Power	$x^n = x$ written n times; $x^0 = \epsilon$	$x^3 = ababab$
Reversal	x^R is x written backwards	$x^R = ba$
Prefix	String obtained by removing zero or more symbols from the end	Prefixes of abc : ϵ, a, ab, abc
Suffix	String obtained by removing zero or more symbols from the front	Suffixes of abc : ϵ, c, bc, abc
Substring	Any contiguous part of the string	Substrings of abc include b, bc, ab

A string of length n has exactly **$n + 1$ prefixes**, $n + 1$ suffixes and, if all its symbols are distinct, $n(n + 1)/2 + 1$ distinct substrings (counting ϵ).

1.1.3 Definition of a Language

Definition: Language

A **language** L over an alphabet Σ is any **subset of Σ^*** , that is, $L \subseteq \Sigma^*$. A language is simply a set of strings; each string in it is a *sentence* or *word* of the language.

This definition is deliberately general. It covers a natural language such as English (a set of valid sentences), a programming language such as C (the set of all valid programs), and also artificial sets such as "all binary strings with an even number of 1s". TOC studies how such sets can be *described finitely* and what is needed to *recognise* their members.

Special languages

- **Empty language $\emptyset$** : contains no string at all. Note $\emptyset \neq \{\epsilon\}$. $\{\epsilon\}$ is a language that contains exactly one string, the empty string.
- **Universal language Σ^*** : contains every possible string over Σ .
- **Finite language**: contains a finite number of strings, e.g. $\{aa, ab, ba, bb\}$.
- **Infinite language**: contains infinitely many strings, e.g. all strings of a 's: $\{\epsilon, a, aa, aaa, \dots\}$.

Examples of languages over $\Sigma = \{a, b\}$

Language	Description	Some members
$L_1 = \{w : w = 2\}$	All strings of length exactly 2 (finite)	aa, ab, ba, bb
$L_2 = \{a^n : n \geq 0\}$	Any number of a 's (infinite)	ϵ, a, aa, aaa
$L_3 = \{a^n b^n : n \geq 0\}$	n a 's followed by exactly n b 's	$\epsilon, ab, aabb, aaabbb$
$L_4 = \{w : w = w^R\}$	Palindromes	$\epsilon, a, b, aa, aba, abba$
$L_5 = \{w : w \text{ ends with } ab\}$	Strings ending in ab	$ab, aab, bab, aaab$
$L_6 = \{a^n b^n c^n : n \geq 1\}$	Equal numbers of a, b, c (over $\Sigma = \{a, b, c\}$)	$abc, aabbcc, aaabbbccc$

1.1.4 Operations on Languages

Since languages are sets, all set operations apply. Besides them, there are operations designed specially for sets of strings. Let $L_1 = \{0, 01\}$ and $L_2 = \{1, 11\}$ over $\Sigma = \{0, 1\}$.

Operation	Definition	Example
Union	$L_1 \cup L_2 = \{w : w \in L_1 \text{ or } w \in L_2\}$	$\{0, 01, 1, 11\}$
Intersection	$L_1 \cap L_2 = \{w : w \in L_1 \text{ and } w \in L_2\}$	$\emptyset$ (no common string)
Difference	$L_1 - L_2 = \text{strings in } L_1 \text{ but not in } L_2$	$\{0, 01\}$
Complement	$L' = \Sigma^* - L$ (all strings not in L)	Everything except 0 and 01
Concatenation	$L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$	$\{01, 011, 0111\}$ ($0 \cdot 1, 0 \cdot 11, 01 \cdot 1 = 011, 01 \cdot 11$)
Kleene star	$L^* = L^0 \cup L^1 \cup L^2 \cup \dots$ (zero or more strings of L joined)	For $L = \{ab\}$: $\{\epsilon, ab, abab, ababab, \dots\}$
Positive closure	$L^+ = L^1 \cup L^2 \cup \dots$ (one or more); $L^+ = LL^*$	For $L = \{ab\}$: $\{ab, abab, \dots\}$
Reversal	$L^R = \{w^R : w \in L\}$	$L_1^R = \{0, 10\}$

Useful facts

- $L\emptyset = \emptyset L = \emptyset$ (concatenation with the empty language gives the empty language).
- $L\{\epsilon\} = \{\epsilon\}L = L$ (the language $\{\epsilon\}$ acts as the identity for concatenation).
- $\epsilon \in L^*$ always, even when $L = \emptyset$; that is, $\emptyset^* = \{\epsilon\}$.
- Concatenation of languages is associative but **not** commutative in general.

1.1.5 Ways of Specifying a Language

Method	Idea	Example
Listing	Write all strings (only for finite languages)	$\{00, 01, 10, 11\}$
Set-builder notation	State a property that members satisfy	$\{w \in \{0,1\}^* : w \text{ has an even number of 1s}\}$
Regular expression	Compact algebraic formula	$(a + b)^*ab$ describes strings ending in ab
Grammar	Set of rewriting rules that generate the strings	$S \rightarrow aSb \mid \epsilon$ generates $a^n b^n$
Automaton	Machine that accepts exactly the strings of the language	DFA of Figure 1.4

1.1.6 The Central Problem of the Subject

For a given language L the **membership problem** asks: *given a string w , is $w \in L$?* Much of TOC is the study of how hard this question is for different kinds of languages. For some languages a very simple machine answers it; for others, only a more powerful machine can, and for some no machine can do it at all.

A deep observation explains why this matters. Σ^* is countably infinite, but the set of **all languages** over Σ (the power set of Σ^*) is **uncountable**. Every machine, program or grammar, however, is a finite string, so there are only countably many of them. Hence **there are far more languages than programs**: most languages cannot be described or recognised by any algorithm. This is the first hint of the limits of computation.

Poly Notes Hub

1.2 Definition of Grammars

1.2.1 Why Grammars?

Listing the strings of an infinite language is impossible, so we need a finite way to *generate* them. A **grammar** does this with a finite set of rewriting rules (*productions*). Starting from a special start symbol and applying rules repeatedly, we can produce every string of the language and nothing else. This is exactly how the grammar of English tells us how a sentence is built from a noun phrase and a verb phrase, and how the grammar of C tells us how a statement is built from expressions.

1.2.2 Formal Definition

Definition: Grammar

A grammar is a 4-tuple $G = (V, T, P, S)$ where:

- V is a finite set of **variables** (non-terminals or syntactic categories);
- T is a finite set of **terminals** (the symbols of the alphabet), with $V \cap T = \emptyset$;
- P is a finite set of **productions** (rules) of the form $\alpha \rightarrow \beta$, where $\alpha \in (V \cup T)^+$ contains at least one variable and $\beta \in (V \cup T)^*$;
- $S \in V$ is the **start symbol**.

Component	Role	Notation convention
Variables V	Placeholders that are eventually replaced	Capital letters: A, B, S
Terminals T	Actual symbols appearing in final strings	Small letters or digits: $a, b, 0, 1$
Productions P	Replacement rules; $A \rightarrow \alpha \mid \beta$ means $A \rightarrow \alpha$ or $A \rightarrow \beta$	$\alpha \rightarrow \beta$
Start symbol S	Symbol from which every derivation begins	S

1.2.3 Derivations and the Language of a Grammar

- **Direct derivation ($\Rightarrow$)**: if $\alpha \rightarrow \beta$ is a production, then $x\alpha y \Rightarrow x\beta y$ (one rule applied once).
- **Derivation ($\Rightarrow^*$)**: zero or more direct derivation steps, e.g. $S \Rightarrow^* w$.
- **Sentential form**: any string of variables and terminals derivable from S .
- **Sentence**: a sentential form that contains only terminals.
- **Leftmost / rightmost derivation**: at every step the leftmost / rightmost variable is replaced.

Definition: Language generated by a grammar

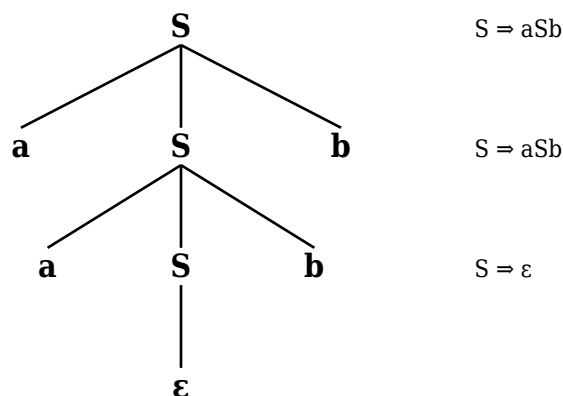
$L(G) = \{ w \in T^* : S \Rightarrow^* w \}$. That is, $L(G)$ is the set of all terminal strings derivable from the start symbol.

Example 1: Grammar for $a^n b^n$, $n \geq 0$

$G = (\{S\}, \{a, b\}, P, S)$ with productions $S \rightarrow aSb \mid \epsilon$

Derivation of **aaabbb**:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbbb \Rightarrow aaabbb \quad (\text{the last step applies } S \rightarrow \epsilon)$$



Yield (read leaves left to right): $a a \epsilon b b = aabb$

Figure 1.2: Derivation (parse) tree of $aabb$ using $S \rightarrow aSb \mid \epsilon$

Example 2: Grammar for a^*

Productions: $S \rightarrow aS \mid \epsilon$. Derivation of aaa : $S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaa$.

Example 3: Grammar for palindromes over $\{a, b\}$

Productions: $S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$. Derivation of $abbba$: $S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbba$ (using $S \rightarrow b$ in the last step).

Example 4: Grammar for $a^n b^n c^n$, $n \geq 1$ (context-sensitive)

Productions: $S \rightarrow aSBC \mid aBC$; $CB \rightarrow BC$; $aB \rightarrow ab$; $bB \rightarrow bb$; $bC \rightarrow bc$; $cC \rightarrow cc$.

Derivation of $aabbcc$:

$$S = aSBC \Rightarrow aaBCBC \Rightarrow aaBBCC \Rightarrow aabBCC \Rightarrow aabbCC \Rightarrow aabbcC \Rightarrow aabbcc$$

1.2.4 Chomsky Hierarchy

In 1956 Noam Chomsky classified grammars into four types by restricting the form of productions. Each type generates a strictly larger class of languages than the type below it, and each class corresponds to a particular kind of machine.

Type	Grammar	Production form	Language class	Accepting machine
3	Regular	$A \rightarrow aB$ or $A \rightarrow a$ (right-linear)	Regular	Finite Automaton (FA)
2	Context-Free (CFG)	$A \rightarrow \gamma$ (single variable on the left)	Context-Free (CFL)	Pushdown Automaton (PDA)
1	Context-Sensitive (CSG)	$\alpha A \beta \rightarrow \alpha \gamma \beta$, $\gamma \neq \epsilon$ ($ \alpha \leq \beta $)	Context-Sensitive (CSL)	Linear Bounded Automaton (LBA)
0	Unrestricted	$\alpha \rightarrow \beta$ (α has at least one variable)	Recursively Enumerable	Turing Machine (TM)

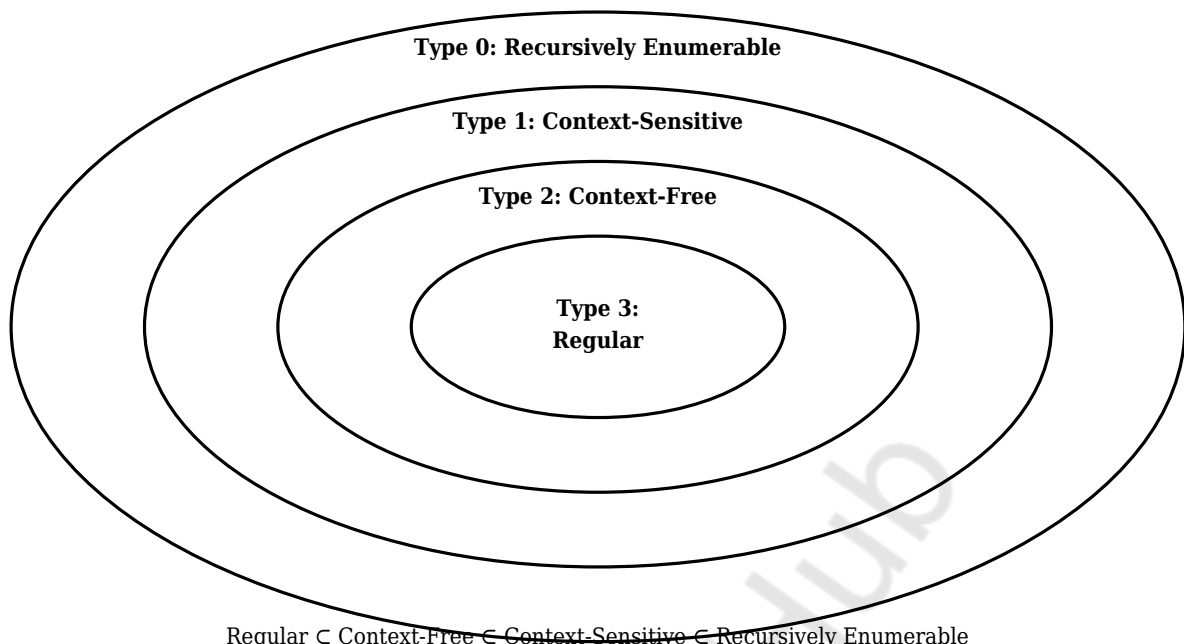


Figure 1.3: Chomsky hierarchy: each class properly contains the one inside it

Class	Typical example language	Why it fits this class
Regular	$\{w : w \text{ ends with } 01\}, a^*b^*$	Needs only finite memory (no counting without bound)
Context-Free	$\{a^n b^n : n \geq 0\}$, palindromes, balanced brackets	Needs a stack to match or count
Context-Sensitive	$\{a^n b^n c^n : n \geq 1\}$	Needs to compare three counts together
Recursively Enumerable	Language of the halting problem	Machine may run forever on non-members

1.2.5 Key Observations

- The hierarchy is a **proper** containment: $\text{Type } 3 \subset \text{Type } 2 \subset \text{Type } 1 \subset \text{Type } 0$.
- A language is called regular / context-free / etc. if *some* grammar of that type generates it.
- Two different grammars can generate the same language; they are called **equivalent** grammars.
- A CFG is **ambiguous** if some string has two different parse trees. Ambiguity is undesirable in programming-language grammars because the meaning of a program would not be unique.

1.3 Definition of Automata

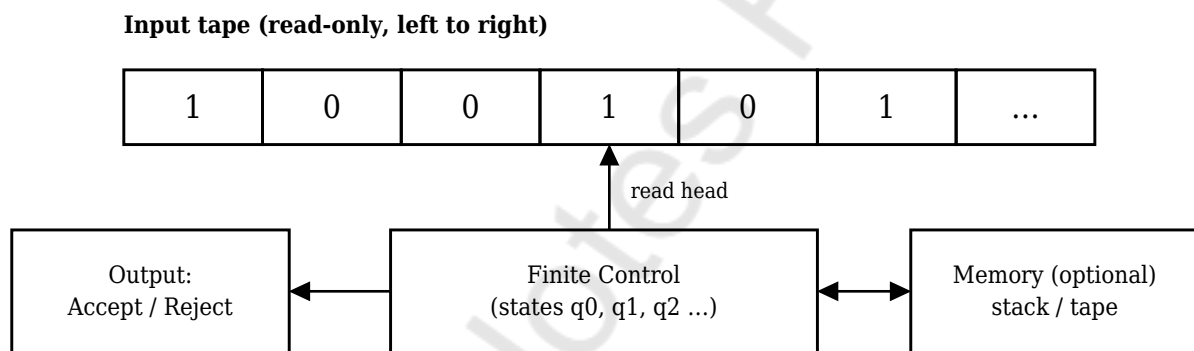
1.3.1 What is an Automaton?

An **automaton** (plural *automata*) is an abstract, self-operating mathematical machine that reads an input string symbol by symbol, changes its internal state according to fixed rules, and finally decides whether to **accept** or **reject** the string. The set of all strings it accepts is the language **recognised** by the automaton. Where a grammar *generates* a language, an automaton *recognises* it.

Informal Definition

An automaton is a machine with a finite set of states, an input tape, a transition rule and a set of accepting states. Its job is to decide membership: for an input w , is w in the language?

1.3.2 Components of an Automaton



Every automaton = input + finite control + (optional) memory + decision

Figure 1.4: General model of an automaton

Component	Function
Input tape	Holds the input string; divided into cells, each holding one symbol.
Read head	Points to the current symbol and moves (usually left to right) along the tape.
Finite control	The set of states and the transition rules; remembers a bounded amount of information.
Memory (optional)	Extra storage: none for FA, a stack for PDA, a tape for LBA / TM.
Output	Decision: accept or reject (an <i>acceptor</i>), or a produced string (a <i>transducer</i>).

1.3.3 Types of Automata

Automaton	Memory	Language class accepted	Example it can accept	Example it cannot accept
Finite Automaton (DFA / NFA)	None (only states)	Regular	Strings ending in 01	$\{a^n b^n\}$
Pushdown Automaton (PDA)	One stack (LIFO)	Context-Free	$\{a^n b^n\}$, palindromes	$\{a^n b^n c^n\}$
Linear Bounded Automaton	Tape limited to input length	Context-Sensitive	$\{a^n b^n c^n\}$	Halting-problem language
Turing Machine	Unbounded tape, read and write	Recursively Enumerable	Anything computable by an algorithm	Uncomputable problems

Deterministic and non-deterministic automata

- **Deterministic:** from each state and input symbol there is *exactly one* next move.
- **Non-deterministic:** there may be several (or no) possible moves; the string is accepted if *at least one* sequence of moves leads to acceptance.
- For finite automata, the two have **equal power**: every NFA can be converted into a DFA (subset construction), though the DFA may need up to 2^n states for an n -state NFA.
- For pushdown automata, non-determinism adds power: DPDA $\subset$ NPDA. For Turing machines, the two are equivalent.

1.3.4 Formal Definition of a DFA

Definition: Deterministic Finite Automaton (DFA)

A DFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states;
- Σ is the finite input alphabet;
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function;
- $q_0 \in Q$ is the single start (initial) state;
- $F \subseteq Q$ is the set of final (accepting) states.

The extended function δ^* processes whole strings: $\delta^*(q, \epsilon) = q$ and $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$.

$$L(M) = \{ w \in \Sigma^* : \delta^*(q_0, w) \in F \}$$

Notation used in transition diagrams

- A **circle** represents a state; an **arrow with a label** represents a transition on that symbol.
- An arrow coming from nowhere marks the **start state**.
- A **double circle** marks a **final (accepting) state**.
- A transition from a state back to itself is called a **self-loop**.

Example: DFA accepting all binary strings that end with 01

$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$. The states record progress: q_0 = last symbol is not a useful 0, q_1 = just saw 0, q_2 = just saw 01.

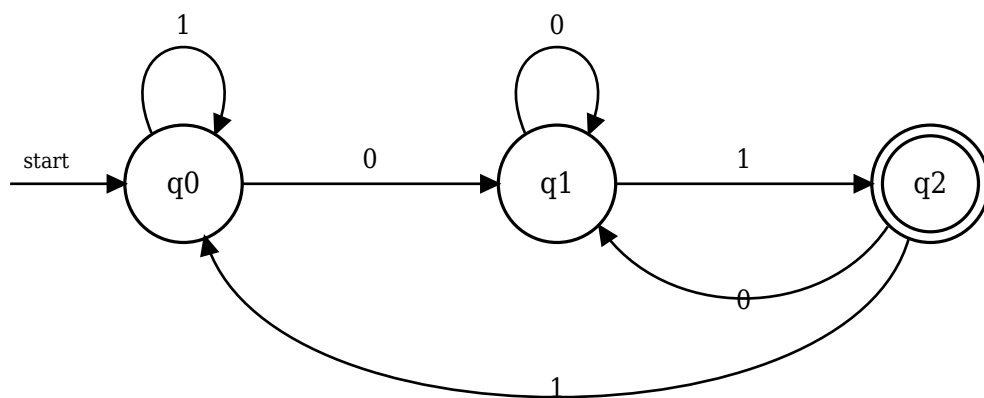


Figure 1.5: Transition diagram of a DFA that accepts strings ending with 01

State	On input 0	On input 1
$\rightarrow q_0$ (start)	q_1	q_0
q_1	q_1	q_2
$* q_2$ (final)	q_1	q_0

Trace for 1001: $q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_1 \xrightarrow{1} q_2$. The machine stops in a final state, so 1001 is **accepted**.

Trace for 110: $q_0 \xrightarrow{1} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1$. q_1 is not final, so 110 is **rejected**.

1.3.5 Grammars versus Automata

Aspect	Grammar	Automaton
Purpose	Generates the strings of a language	Recognises (accepts) the strings of a language
Viewpoint	Top-down: start symbol to string	Bottom-up: string to accept / reject decision
Description	Variables, terminals, productions, start symbol	States, alphabet, transitions, start and final states
Regular languages	Regular (Type 3) grammar	Finite automaton
Context-free languages	Context-free (Type 2) grammar	Pushdown automaton
Recursively enumerable	Unrestricted (Type 0) grammar	Turing machine

The equivalence of each grammar type with its machine is the unifying result of the subject: **language, grammar and automaton are three views of the same object.**

1.4 Some Applications

TOC may look abstract, but its models are present in almost every piece of software and hardware. The table below lists the main application areas; each is explained after it.

Application area	Model used	Purpose
Lexical analysis (compiler)	Regular expressions, DFA	Split source code into tokens: keywords, identifiers, numbers
Syntax analysis (parser)	Context-free grammars, PDA	Check that the token sequence follows the language grammar
Pattern search and text processing	Regular expressions, NFA / DFA	grep, find-and-replace, input validation
Digital circuit design	Finite state machines (Mealy / Moore)	Counters, sequence detectors, controllers
Protocol and software verification	Finite-state models, model checking	Prove that a protocol never reaches a bad state
Document formats	Context-free grammars	Validate HTML / XML / JSON structure
Natural language processing	Grammars, parsers, automata	Sentence parsing, speech recognition, spell checkers
Embedded and game control	State machines	Traffic lights, vending machines, elevators, game character AI
Computability and complexity	Turing machines	Know which problems are unsolvable or intractable
Bioinformatics	Finite automata, grammars	DNA / protein pattern matching

1.4.1 Compiler Design

A compiler translates a high-level program into machine code in several phases, and TOC supplies the theory for the first two.

- **Lexical analyser (scanner):** tokens are described by regular expressions, e.g. an identifier is `letter(letter | digit)*`. Tools such as *Lex / Flex* convert these expressions into a DFA that scans the program in one pass.
- **Parser:** the structure of the language is described by a context-free grammar, e.g. $E \rightarrow E + T \mid T, T \rightarrow T * F \mid F, F \rightarrow (E) \mid \text{id}$. Tools such as *Yacc / Bison* build a pushdown-automaton-based parser from it.
- Syntax errors such as a missing bracket are detected exactly when the parser cannot continue a derivation.

1.4.2 Pattern Matching and Text Editors

Searching with wildcards, validating an e-mail address or a mobile number in a web form, and the *grep* command all use regular expressions internally. The expression is converted to an NFA or DFA, and the text is fed through it. A DFA gives search time proportional to the length of the text.

1.4.3 Hardware and Embedded Systems

- Sequential circuits (counters, registers, sequence detectors) are finite state machines implemented with flip-flops.
- A **traffic-light controller** has states red, green and yellow, with timer events as inputs.
- A **vending machine** has states for the amount of money inserted; coin insertion is the input and delivering the product is the output.

1.4.4 Verification and Software Engineering

Communication protocols, operating-system schedulers and safety-critical controllers are modelled as finite-state systems, and **model checkers** explore all reachable states to prove that no unsafe state can occur. UML state diagrams used in software design are an extension of the same idea.

1.4.5 Computability and Complexity

- **Halting problem:** Turing proved that no algorithm can decide for every program and input whether the program eventually stops. Consequently no tool can find all infinite loops or all bugs automatically.
- **Decidable and undecidable problems** tell engineers when to stop searching for a perfect algorithm and settle for approximations.
- **P versus NP:** problems in P are solvable in polynomial time; NP-complete problems (such as the travelling salesman or satisfiability) have no known efficient algorithm. Modern cryptography relies on problems believed to be hard.

1.4.6 Summary of the Chapter

- An **alphabet** Σ is a finite set of symbols; Σ^* is the set of all strings; a **language** is any subset of Σ^* .
- A **grammar** $G = (V, T, P, S)$ generates a language using productions; $L(G) = \{w \in T^* : S \Rightarrow^* w\}$.
- The **Chomsky hierarchy** has four types: Regular $\subset$ Context-Free $\subset$ Context-Sensitive $\subset$ Recursively Enumerable.
- An **automaton** accepts a language; a DFA is the 5-tuple $(Q, \Sigma, \delta, q_0, F)$.
- Machines in increasing power: FA, PDA, LBA, Turing machine, matching the four grammar types.
- Applications: compilers, pattern matching, circuits, protocols, verification, and understanding the limits of computing.

Practice Section A: 15 Multiple Choice Questions

Choose the correct option. The answer key with short explanations is given at the end of this section.

- Q1.** Which of the following can NOT be an alphabet?
- (A) $\{0, 1\}$
 - (B) $\{a, b, c\}$
 - (C) The set of all natural numbers
 - (D) $\{x\}$
- Q2.** The length of the empty string ϵ is:
- (A) 0
 - (B) 1
 - (C) Undefined
 - (D) Infinity
- Q3.** If $\Sigma = \{a, b\}$, how many strings of length 3 exist over Σ ?
- (A) 3
 - (B) 6
 - (C) 9
 - (D) 8
- Q4.** Σ^* (Kleene closure of Σ) contains:
- (A) Only non-empty strings
 - (B) All strings over Σ including ϵ
 - (C) Only strings of length 1
 - (D) Only finite languages
- Q5.** Let $L_1 = \{0, 01\}$ and $L_2 = \{1, 11\}$. The number of strings in $L_1 L_2$ is:
- (A) 2
 - (B) 4
 - (C) 3
 - (D) 6
- Q6.** For $L = \{ab\}$, which of the following strings does NOT belong to L^* ?
- (A) ϵ
 - (B) ab
 - (C) $abab$
 - (D) aba
- Q7.** In the grammar $G = (V, T, P, S)$, the symbol T stands for:
- (A) Transitions
 - (B) Terminals
 - (C) Tokens
 - (D) Tape symbols

- Q8.** The grammar $S \rightarrow aSb \mid \epsilon$ generates the language:
- (A) $\{a^n b^n : n \geq 0\}$
 - (B) $\{a^n b^m : n, m \geq 0\}$
 - (C) a^*b^*
 - (D) $(ab)^*$
- Q9.** A grammar whose productions have the form $A \rightarrow \gamma$ (single variable on the left side) is called:
- (A) Regular
 - (B) Context-sensitive
 - (C) Context-free
 - (D) Unrestricted
- Q10.** Which machine accepts exactly the regular languages?
- (A) Pushdown automaton
 - (B) Finite automaton
 - (C) Turing machine
 - (D) Linear bounded automaton
- Q11.** Which automaton uses a stack as its memory?
- (A) DFA
 - (B) NFA
 - (C) Turing machine
 - (D) Pushdown automaton
- Q12.** A DFA is formally defined as a 5-tuple:
- (A) $(Q, \Sigma, \delta, q_0, F)$
 - (B) (V, T, P, S, F)
 - (C) $(Q, \Gamma, \delta, q_0, B)$
 - (D) $(Q, \Sigma, \Gamma, \delta, F)$
- Q13.** Which of the following is the most powerful computational model?
- (A) Finite automaton
 - (B) Pushdown automaton
 - (C) Turing machine
 - (D) Regular expression
- Q14.** The language $\{a^n b^n c^n : n \geq 1\}$ is:
- (A) Regular
 - (B) Context-sensitive but not context-free
 - (C) Context-free
 - (D) Finite
- Q15.** Lexical analysis in a compiler is based mainly on:
- (A) Turing machines
 - (B) Context-free grammars
 - (C) Pushdown automata
 - (D) Regular expressions and finite automata

Answer Key

Q	Ans	Explanation
1	C	An alphabet must be finite; the natural numbers form an infinite set.
2	A	$ \epsilon = 0$ as it has no symbols.
3	D	$2^3 = 8$.
4	B	Σ^* is the set of all strings of every length including ϵ ; Σ^+ excludes ϵ .
5	C	01, 011, 011, 0111; duplicates merge, giving $\{01, 011, 0111\} = 3$ strings.
6	D	L^* contains only repetitions of ab ; aba is not one.
7	B	T is the finite set of terminal symbols.
8	A	Each use of $S \rightarrow aSb$ adds one a on the left and one b on the right, keeping counts equal.
9	C	This is the definition of a Type 2 (context-free) grammar.
10	B	Regular languages (Type 3) correspond to finite automata.
11	D	A PDA is a finite automaton equipped with a stack.
12	A	Q = states, Σ = alphabet, δ = transition function, q_0 = start state, F = final states.
13	C	A Turing machine recognises recursively enumerable languages, the largest class here.
14	B	It needs three equal counts, which a stack cannot track; a linear bounded automaton can.
15	D	Tokens are specified by regular expressions and recognised by DFAs.

Quick key: 1-C, 2-A, 3-D, 4-B, 5-C, 6-D, 7-B, 8-A, 9-C, 10-B, 11-D, 12-A, 13-C, 14-B, 15-D

Practice Section B: 10 Broad Questions

Attempt each answer in about 1.5 to 2 pages in an examination style, using definitions, examples and diagrams wherever useful.

- Q1.** Define alphabet, string, length of a string and language. Explain the terms Σ^* , Σ^+ and ε with suitable examples. How many strings of length n exist over an alphabet with k symbols?
- Q2.** Explain all the operations on languages (union, intersection, complement, concatenation, Kleene star and positive closure) with examples. Show that $L\{\varepsilon\} = L$ and $L\emptyset = \emptyset$.
- Q3.** Give the formal definition of a grammar. Explain the terms variable, terminal, production, start symbol, derivation, sentential form and the language generated by a grammar.
- Q4.** Construct context-free grammars for (a) $\{a^n b^n : n \geq 1\}$ and (b) all palindromes over $\{a, b\}$. Derive the strings $aaabbb$ and $abbba$ using your grammars and draw the parse tree for one of them.
- Q5.** Describe the Chomsky hierarchy in detail. Give the production form, language class, accepting machine and one example for each type, and draw the diagram showing the containment of the classes.
- Q6.** What is an automaton? Draw and explain the general model of an automaton with its components. List the main types of automata and the language class each accepts.
- Q7.** Define a DFA formally. Design a DFA over $\{0, 1\}$ that accepts all strings ending with 01. Draw its transition diagram, give the transition table and show the trace for the input 1001.
- Q8.** Differentiate between: (a) deterministic and non-deterministic automata, (b) a grammar and an automaton, (c) a finite language and an infinite language, with examples.
- Q9.** Explain with examples the applications of the Theory of Computation in compiler design, pattern matching, hardware design and software verification. Mention the role of regular expressions and context-free grammars.
- Q10.** Explain the relationship between languages, grammars and automata. Why are there languages that no machine can recognise? Briefly introduce the idea of decidability and the halting problem.

End of Chapter 1