CS2009 Theory of Computation
Finite Automata & Regular Languages

Dr. Maithilee Laxmanrao Patawar

Department of CSE,
IIITDM Kancheepuram

January 6, 2026

Course details

- Course code: CS2009
- Course name: Theory of Computation
- Credits: 04
- Lectures: 03, Tutorials: 01
- Mark distribution:
  1. Mid term: 25%
  2. Internals: 25%
  3. End term: 50%

Course Objectives

- Provide fundamentals of computing models
  - Finite state automata,
  - Push down automata,
  - Linear bounded automata
  - Turing machine
- Powers and limitations of the above models
- Solvability and Tractability

Course Outcomes

- To design various computational models useful for solving problems
- To understand the relationship among digital computer, algorithm and Turing machine.
- To verify whether a given problem is solvable or tractable.

Recommended Reading

- Introduction to Automata Theory, Languages and Computation, Hopcroft, Motwani, and Ullman, Pearson Publishers, Third Edition, ISBN: 9780321455369, 2006.
- Supplementary reading:
  1. Elements of the Theory of Computation, H. R. Lewis and C.H. Papadimitriou, Prentice Hall Publishers, ISBN. 0-13-2624 78-8, 1981
  2. Introduction to Languages and the Theory of Computation, John. C. Martin, Tata McGraw-Hill, ISBN 978-00731914612003.

Why Study Automata Theory?

- Automata Theory is about studying **machines that follow rules**.
- These are not physical machines like cars or washing machines.
- They are **abstract machines**.

What Do These Machines Do?

Abstract machines in Automata Theory:

- Read input **step by step**
- Change **states**
- Decide **what is valid and what is not**

Why Study Automata Theory?

- Automata Theory studies **abstract machines** and the problems they can solve.
- These machines follow **well-defined rules** to process input.
- It forms a **core foundation** of Computer Science.

Key Question:

What kinds of problems can computers solve, and how efficiently?

What Questions Does Automata Theory Answer?

Automata Theory helps us answer:

- Can this problem be solved by a computer?
- How simple or complex does the solution need to be?
- What kind of machine is sufficient to solve it?

Why Is Automata Theory Important?

Automata Theory helps us to:

- Understand how computers process input step-by-step
- Classify problems based on **computational power**
- Design reliable software and hardware systems
- Build compilers, analyzers, and verification tools

Why Is Automata Theory a Foundation?

Automata Theory forms the foundation of:

- Compilers
- Programming Languages
- Operating Systems
- Computer Networks
- Security Protocols

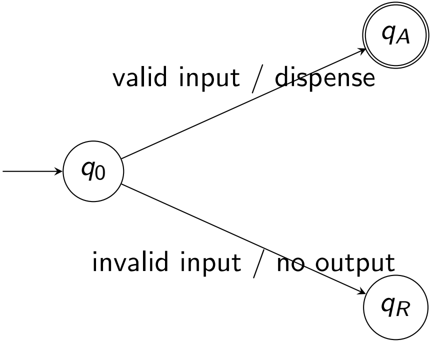
Finite Automaton: Real-Life Analogy

Think of a **vending machine**:

- You insert coins (input)
- The machine moves through states
- When enough money is reached, it accepts and gives you a snack

This behavior is exactly how a Finite Automaton works.

Automata for a Vending Machine



State meaning:

- q_0 : Initial state (checks input)
- q_A : Accepting state (item dispensed)
- q_R : Rejecting state (no output)

Finite Automaton: Intuitive Example

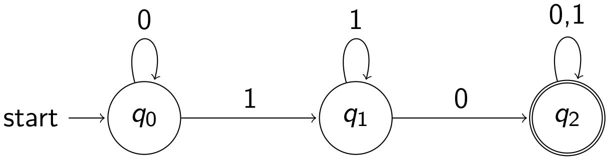
Real-life analogy: Vending Machine

- States represent the amount of money inserted
- Input symbols are coins
- Accepting state means enough money has been inserted

Finite Automata work in the same way:

Read input $\rightarrow$ *change state* $\rightarrow$ *decide YES or NO*

Example DFA: Binary Strings containing a substring 10



This DFA accepts all binary Strings containing a substring 10

Application 1: Digital Circuit Design

Digital circuits have:

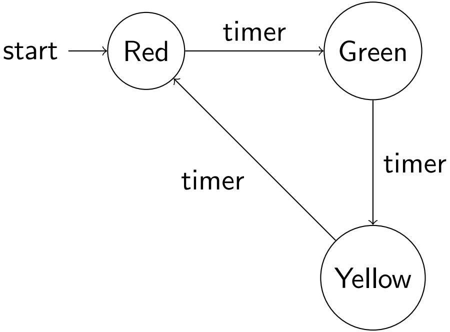
- A finite number of states (ON/OFF)
- Inputs (signals, clock)
- Outputs based on current state

Example: Traffic Light Controller

- States: Red, Yellow, Green
- Transitions based on timer

Such systems can be modeled using **Finite Automata**.

DFA for Traffic Light Controller



Application 2: Lexical Analyzer in Compiler

The **Lexical Analyzer**:

- Is the first phase of a compiler
- Breaks source code into **tokens**

Example:

```
int count = 10;
```

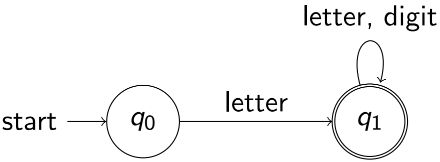
Tokens:

- `int` $\rightarrow$ keyword
- `count` $\rightarrow$ identifier
- 10 $\rightarrow$ number

DFA for Identifiers

Identifiers follow the pattern:

Letter followed by letters or digits



Application 3: Text and Pattern Searching

Used in:

- Search engines
- Text editors (Ctrl + F)
- Command-line tools like `grep`

These systems:

- Scan text character by character
- Match predefined patterns

Finite Automata enable **efficient pattern matching**.

Application 4: System Verification

Many systems have:

- Finite number of states
- Critical safety requirements

Examples:

- Communication protocols
- Authentication systems
- Secure data exchange protocols

Finite Automata help ensure:

- No unsafe states are reached
- Correct execution flow

Why Finite States Enable Verification

A system is suitable for automata-based verification if:

- Its behavior can be described using a **finite set of states**
- Transitions between states depend only on the current state and input

Key Idea: If all possible states and transitions are known, the system can be **exhaustively analyzed**.

Examples of Systems with Finite States

Common systems that naturally have finite states include:

- **Communication Protocols**
  States such as *waiting*, *sending*, *receiving*, *error*
- **Authentication Systems**
  States such as *logged out*, *login attempt*, *authenticated*, *blocked*
- **Secure Data Exchange Protocols**
  States such as *key generation*, *key exchange*, *encrypted communication*

Why Verification Is Necessary

Without systematic verification:

- A system may enter an **unsafe or unintended state**
- Certain sequences of inputs may lead to **deadlock or failure**
- Security-sensitive systems may become **vulnerable to attacks**

Testing alone cannot cover all possible execution paths.

What Finite Automata Help Ensure

Using automata-based verification, we can ensure:

- **Safety:** Unsafe states are never reachable
- **Correctness:** The system follows the intended execution flow
- **Completeness:** All valid behaviors are allowed

These guarantees are especially important in security-critical systems.

Try some examples

Design an automata for

- a switch (On/off actions)
- accept the word "ToC"

Summary

- Automata Theory studies abstract computational machines
- Finite Automata are the simplest and most practical models
- Widely used in hardware, software, and verification
- Forms the foundation of compilers, search tools, and protocols

Understanding Automata = Understanding Computation

Real life examples

1. **Online Exam Access**: A student can start the exam only once. Refreshing the page does not restart the exam.
2. **Automatic Email Filter**: Emails are marked as spam if certain keywords are detected; otherwise, they go to inbox.
3. **USB Device Connection**: A USB device is either connected or disconnected. Data transfer happens only when connected.
4. **Parking Gate Barrier**: The gate opens when a valid ticket is scanned and closes after the vehicle passes.

Where We Are Now

So far, we have:

- Designed automata for real-life systems
- Seen how machines change states based on input

Next question:

What exactly do these machines read and accept?

The central Concept of Automata Theory

- Alphabets
- Strings
- Languages

Alphabet

An **alphabet**, denoted by Σ, is a:

- Finite
- Non-empty
- Set of symbols

Important: Symbols do not have to be letters or digits.

Examples of Alphabets

- Binary strings: $\Sigma = \{0, 1\}$
- Online exam system: $\Sigma = \{\text{start}, \text{refresh}\}$
- USB device: $\Sigma = \{\text{plugIn}, \text{unplug}, \text{transfer}\}$

Strings

A **string** is a:

- Finite sequence of symbols
- Each symbol must come from the alphabet

Examples over $\Sigma = \{0, 1\}$:

- ϵ (empty string)
- 0, 1
- 01, 1101

Empty String

- The empty string is denoted by ϵ
- It contains no symbols
- Length of ϵ is 0

Meaning: The machine receives no input.

Valid and Invalid Strings

Once an automaton is fixed:

- A **valid string** is accepted by the automaton
- An **invalid string** is rejected by the automaton

The automaton decides validity, not us.

Example: USB Device Automaton

Valid strings:

- plugIn
- plugIn transfer
- plugIn transfer transfer

Invalid strings:

- transfer
- unplug
- transfer plugIn

Language

A **language** is a:

- Set of strings
- Defined over an alphabet

Language of an automaton: The set of all strings it accepts.

Example of a Language

For the USB device automaton:

$$L = \{\text{plugIn}, \text{plugIn transfer}, \text{plugIn transfer transfer}, \dots\}$$

This is an infinite language.

Why Do We Need Grammars?

So far, automata:

- Check whether a string is valid

New question:

How can we generate all valid strings of a language?

Grammar

A **grammar**:

- Is a set of rules
- Generates strings of a language

Automata recognize strings, grammars generate strings.

Example Grammar

Grammar for binary strings ending with 1:

$$S \rightarrow 0S \mid 1S \mid 1$$

Starting from S, applying rules produces valid strings.

Regular Expressions: Motivation

Drawing automata can be inconvenient.

Question:

Can we describe the same language in a compact form?

Answer: Regular Expressions.

Regular Expressions

A **regular expression**:

- Describes a regular language
- Uses symbols, concatenation, union, and repetition

Basic operators:

- $|$ (OR)
- $*$ (zero or more repetitions)

Regular Expression Examples

- Binary strings ending with 1:
  $(0|1)^*1$
- USB valid transfers:
  $\text{plugIn}(\text{transfer})^*$
- Identifier:
  $\text{letter}(\text{letter}|\text{digit})^*$

Unifying Everything

The same language can be described using:

- Finite Automata
- Grammars
- Regular Expressions

They are different views of the same concept.

Summary

- Alphabet defines symbols
- Strings are sequences of symbols
- Language is a set of valid strings
- DFA recognizes a language
- Grammar generates a language
- Regular Expression describes a language

Same idea — different representations

Q1: Alphabet vs String

Let $\Sigma = \{0, 1\}$.
Which of the following are **alphabets** and which are **strings**?

- $\{0, 1\}$
- 01
- $\{01\}$
- $0, 1$
- ϵ

Q2: Symbols vs Characters

Let $\Sigma = \{\text{start}, \text{refresh}\}$.
Which of the following are valid **symbols** in Σ?

- start
- refresh
- s
- start refresh
- ϵ

Q3: String or Not?

Let $\Sigma = \{a, b\}$.
Which of the following are **strings over** Σ?

- abba
- aabb
- abc
- ϵ
- {a,b}

Q4: Valid vs Invalid Strings

Consider a DFA that accepts **all binary strings ending with 1**.
Classify the following strings as **valid** or **invalid**.

- 1
- 01
- 10
- 111
- ϵ

Q5: Language vs Alphabet

Which of the following are **languages**?

- $\{0, 1\}$
- $\{0, 1, 01, 10\}$
- $\{\epsilon\}$
- All binary strings of even length
- 01

Q6: Empty String vs Empty Language

Choose the correct statements.

- ϵ is a string
- ϵ is a language
- $\{\epsilon\}$ is a language
- $\emptyset$ is a string
- $\emptyset$ is a language

Q7: Finite vs Infinite Language

State whether each language is **finite** or **infinite**.

- All strings over $\{0, 1\}$
- All binary strings of length exactly 2
- $\{\epsilon\}$
- All strings ending with 1

Q8: Grammar vs Language

Which of the following **define a language**?

- A DFA
- A grammar
- A regular expression
- A string
- An alphabet

Q9: Strings and Membership

Let $\Sigma = \{a, b\}$ and $L = \{\text{all strings that start with } a\}$.
Which strings belong to L?

- a
- ab
- ba
- aaab
- ϵ

Q10: True or False (Explain)

Answer True or False.

- Every string over an alphabet belongs to a language.
- A language can be empty.
- A valid string must be finite.
- An alphabet can be infinite.
- Two different DFAs can define the same language.

Answers: Q1 (Alphabet vs String)

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

- $\{0, 1\}$ **Alphabet**
- 01 **String**
- $\{01\}$ **Alphabet** (set with one symbol)
- $0, 1$ **Not a formal object**
- ϵ **String (empty string)**

Answers: Q2 (Symbols vs Characters)

Let $\Sigma = \{\text{start}, \text{refresh}\}$.

- start **Valid symbol**
- refresh **Valid symbol**
- s **Not a symbol**
- start refresh **String, not a symbol**
- ϵ **String, not a symbol**

Answers: Q3 (String or Not)

Let $\Sigma = \{a, b\}$.

- abba — **String**
- aabb — **String**
- abc — **Not a string** (c not in Σ)
- ϵ — **String**
- {a,b} — **Alphabet, not a string**

Answers: Q4 (Valid vs Invalid Strings)

Language: All binary strings ending with 1.

- 1 **Valid**
- 01 **Valid**
- 10 **Invalid**
- 111 **Valid**
- ϵ **Invalid**

Answers: Q5 (Language vs Alphabet)

- $\{0, 1\}$ **Alphabet (not a language by description)**
- $\{0, 1, 01, 10\}$ **Language**
- $\{\epsilon\}$ **Language**
- All binary strings of even length **Language**
- 01 **String**

Answers: Q6 (Empty String vs Empty Language)

- ϵ is a string **True**
- ϵ is a language **False**
- $\{\epsilon\}$ is a language **True**
- $\emptyset$ is a string **False**
- $\emptyset$ is a language **True**

Answers: Q7 (Finite vs Infinite)

- All strings over $\{0, 1\}$ **Infinite**
- Binary strings of length exactly 2 **Finite**
- $\{\epsilon\}$ **Finite**
- All strings ending with 1 **Infinite**

Answers: Q8 (Grammar vs Language)

Which define a language?

- DFA **Yes**
- Grammar **Yes**
- Regular expression **Yes**
- String **No**
- Alphabet **No**

Answers: Q9 (Membership)

$L = \{\text{strings over } \{a, b\} \text{ that start with } a\}$

- a **Yes**
- ab **Yes**
- ba **No**
- aaab **Yes**
- ϵ **No**

Answers: Q10 (True / False)

- Every string over an alphabet belongs to a language **False**
- A language can be empty **True**
- A valid string must be finite **True**
- An alphabet can be infinite **False**
- Two different DFAs can define the same language **True**