CS2333

Finite Automata I

Hassan Arafat

Computer Science
University of New Brunswick

Source Material

The textbook used in this course is available online on: http://cglab.ca/ michiel/TheoryOfComputation/

The slides and material used in this course are heavily based on, and inspired by material by both Prof. Patricia Evans and Prof. Michael Fleming.

Simple Model of Computation

We consider the decision problem of determining if a string is a member of a certain language. For certain languages not too many resources are required.
A simple automaton has limited memory requirement, and as such would only have a finite set of states. In other words, it only needs to "remember" it's current state.

Finite state automata have lots of uses, especially in applications where memory and complexity are limited:

- hardware circuits
- processing strings (e.g. regular expressions)

Finite Automata

Finite Automata (also referred to as Finite State Machines) are machines that can be represented as a set of states, with transitions between them.
A Finite Automata would read the input one character at a time, and depending on both it's current state and the input character, determine what state to transition to.

The FSM would always start at a certain state, known as the **start state**. When the end of the string is reached, we either **accept** or **reject** based on the current state. The set of states where we accept the string are known as **accept states**

Strings accepted by an automata form a language, languages accepted by Finite Automata are called **regular languages**

Formal Definition

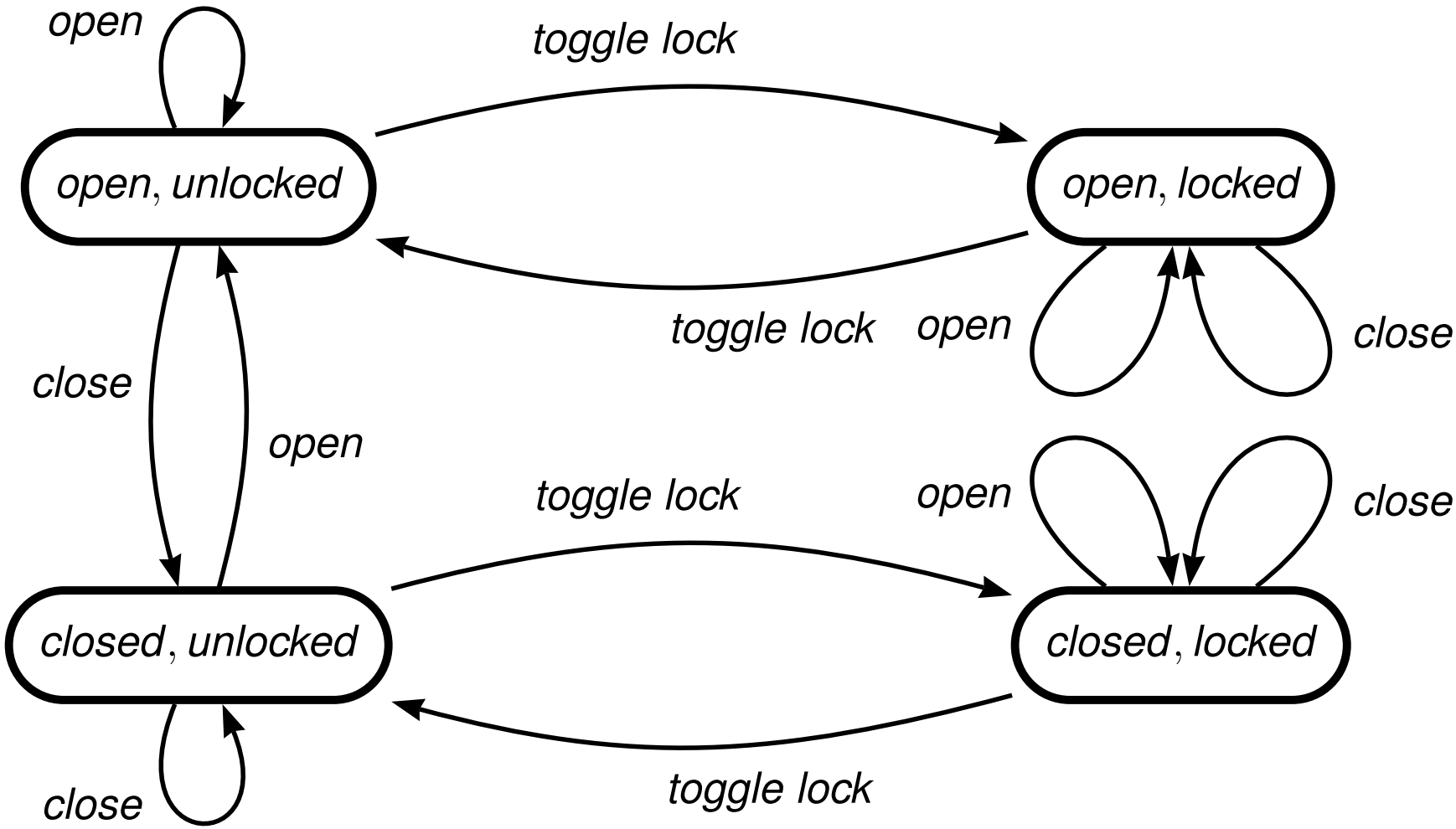
A **finite automaton** is 5-tuple $M = (Q, \Sigma, \delta, q, F)$, where:

- Q is a finite set of **states**
- Σ is a finite set of possible input symbols (**input alphabet**)
- $\delta : Q \times \Sigma \rightarrow Q$ is the **transition function**
  - δ maps every pair of current state in Q and input symbol in Σ to the next state in Q.
- $q \in Q$ is the **start state**
- $F \subseteq Q$ is the set of **accept states**

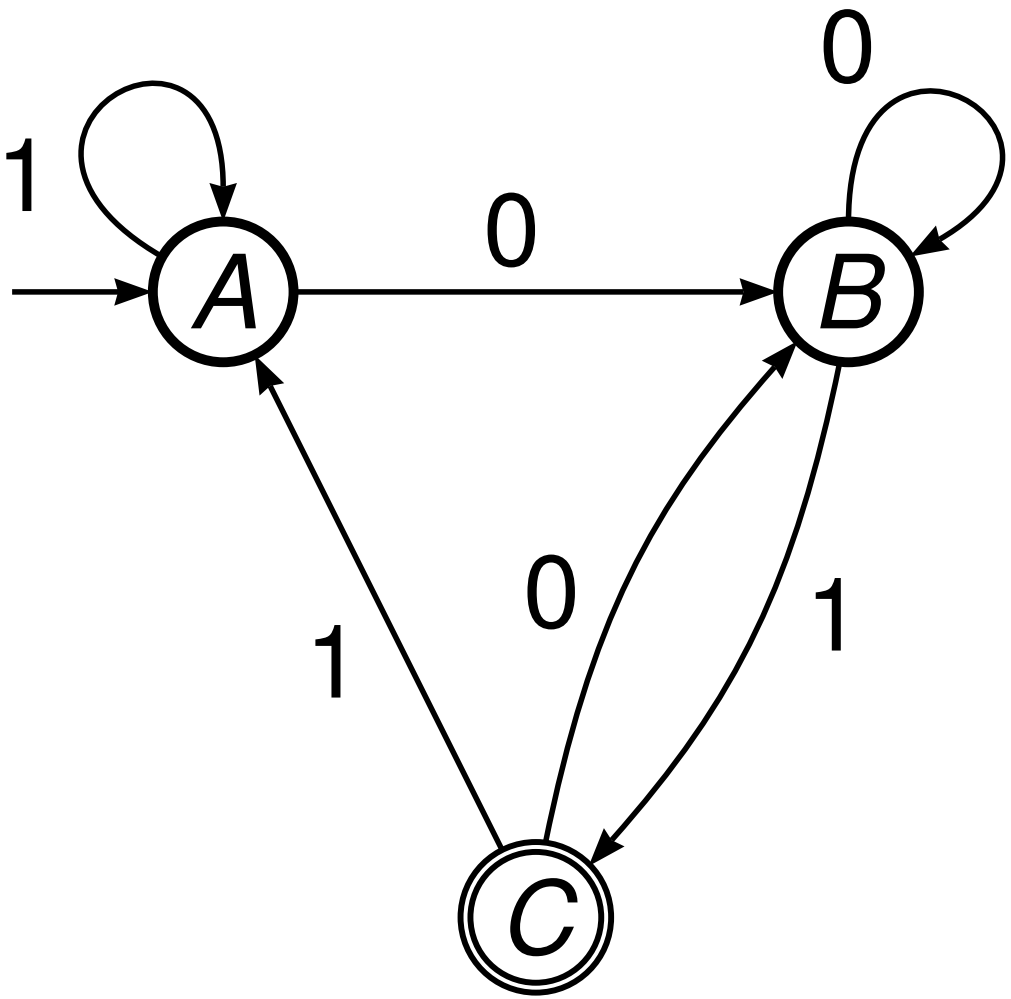
FSM visualization

- Each **state** is denoted by a circle.
- The symbols on each arrow indicate the **input alphabet**.
- Each arrow represents a **transition** from that state when the indicated symbol is read.
- An incoming arrow denotes a **start state**.
- A double circle indicates an **accept state**.

A Door as a Finite Automaton



Navigating an FA



Where does each string lead?

1 1 0

0 1 0 1

0 1 1 0 0

Running a Finite Automaton

We run a finite automaton on a string by:

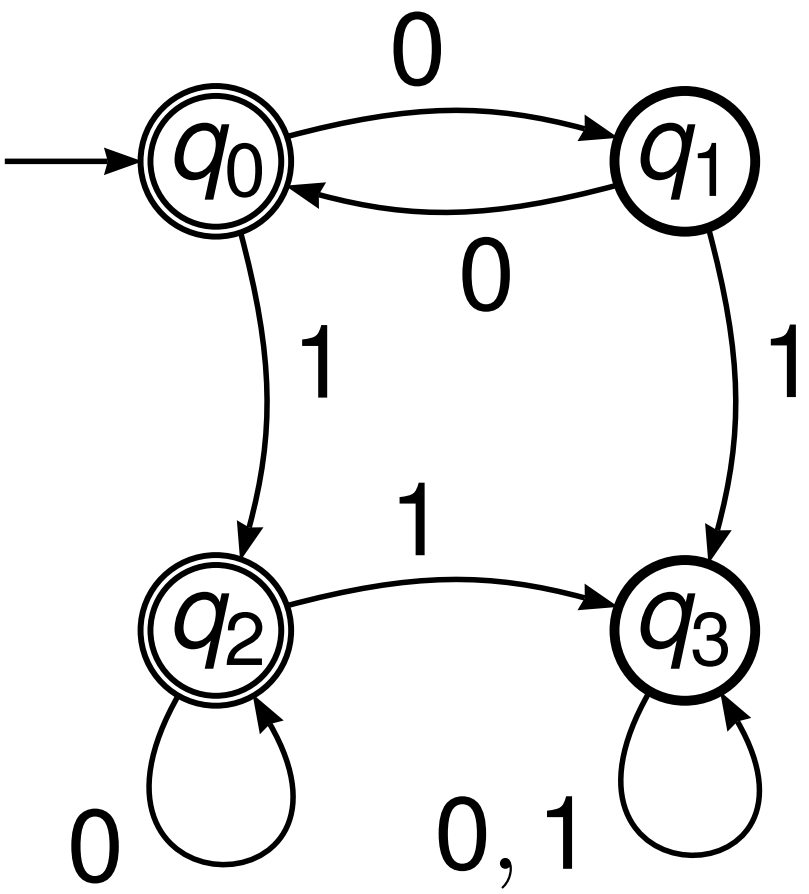
- starting in the start state q_0
- for each symbol in the string, in order:
  - read the symbol
  - use the symbol x and current state q to find next state $\delta(q, x)$
- at the end of the string, accept if and only if our current state $q \in F$

Finite Automaton:Example

Using this, we can, for example, design a simple finite automaton that checks the parity of a binary string, to decide the language

$$\{x \ : \ x \in \{0,1\}^* \text{ and the number of 1s in } x \text{ is even } \}$$

Example Automaton Diagram



Automaton Computation

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton, and
$w = w_1 w_2 \ldots w_n$ be a string over Σ.

Then the sequence of states of this computation is $r_0, r_1, \ldots, r_n$, where

$$r_i = \begin{cases} q_0 & \text{if } i = 0 \\ \delta(r_i, w_{i+1}) & \text{if } i \geq 1 \end{cases}$$

If $r_n \in F$, then M **accepts** w.
If $r_n \notin F$, then M **rejects** w.

Automaton Computation

We can chain together the results of the transition function to get

$$\overline{\delta} : Q \times \Sigma^* \to Q$$

where

$$\overline{\delta}(r, w) = \begin{cases} r & \text{if } w = \varepsilon \\ \delta(\overline{\delta}(r, v), a) & \text{if } w = va, \text{ where } v \in \Sigma^* \text{ and } a \in \Sigma \end{cases}$$

Regular Languages

The **language** of strings accepted by a finite automaton M is

$$L(M) = \{x \in \Sigma^* \mid \overline{\delta}(q_0, x) \in F\}$$

The set, or **class**, of languages accepted by some finite automaton

$$\mathcal{L}(FA) = \{L(M) \mid M \text{ is a finite automaton}\}$$

is the class of **regular languages**.

What Do Finite Automata Accept?

Every automaton partitions the set of all strings Σ^* into $|Q|$ equivalence classes, one class per state.

Each class $Class[p]$, $p \in Q$, is the set of all strings in Σ^* whose computation from q_0 ends at state p.

$$Class[p] = \{x \in \Sigma^* \mid \overline{\delta}(q_0, x) = p\}$$

And $L(M) = \cup_{p \in F} Class[p]$.

We can determine what language an automaton accepts by finding the equivalence class for each state, though sometimes this can be difficult.

We can consider the equivalence classes (and whether certain strings should be equivalent) as we design an automaton.

What Can Finite Automata Accept?

Because Finite Automata can only remember their state, they are limited to accepting languages that require only the following:

- counting **to** a finite number, where you can "remember" the number by the state value.
- counting **modulo** a finite number, where there is a determined number of possible values.
- matching a finite substring or pattern. again, predetermined length.

What Won't Finite Automata Accept?

Since Finite State Automata have a predefined number of states, They cannot:

- count arbitrarily high
- match arbtrarily long strings

Therefore, a Finite Automata can't match languages like:

- Balanced parentheses, $L_C = \{(^i)^j \mid i = j\}$
- $L_P = \{w \in \Sigma^* : w = w^R\}$, where w^R is the reverse of string w
- $L_M = \{ww \mid w \in \Sigma^*\}$

Regular Operations

Let A and B be lanuages. Regular operations on these languages are :

- **Union**: $A \cup B = w \mid w \in A \text{ or } w \in B$
- **Concatenation**: $AB = ww' \mid w \in A \text{ and } w' \in B$
- **Star**: $A* = u_1 u_2 \ldots u_k \mid k \geq 0 \text{ and each } u_i \in A$

Simulations

Given two regular languages A and B we can construct Finite Automata for each of the languages:

- $A \cup B$
- AB
- $A*$
- $B*$

Since a language that can be recognized by a Finite Automata is a regular language, this means that the class of regular languages is **closed** under regular operations.