Finite State Automata

Computing Science @ Aberdeen

Dept. of Computing Science, University of Aberdeen, UK

Languages and Computability (CS3518)

Outline

1. Introduction to the Course
2. Finite State Automata: Preliminaries
3. Finite State Automata: Intuition
4. Transition graph
   - How FSAs Work
5. Formal Definition of FSA
6. How Knowledge on FSAs is Gauged
7. Formal Definition of Computation
8. Designing Finite Automata
9. Final Remarks

Introduction to the Course

Why should we study formal languages and computability?

- To define the **essence of computing**
- To define what an **algorithm** is
- To answer: are there are problems which cannot be solved with computers?
  - These are the **limits of computing**
  - There is no point in waiting for a faster computer/OS
  - Useful knowledge as unsolvable problems are common

How do we study formal languages and computability?

- Using mathematics (sets, functions, relations)
- We strip "computing" back to its bare bones
- No technologies, no hardware, no hype

Preliminaries I

Finite State Automata:

- Also called FSA, finite state machine (FSM) or just automata
  - "Automata" – plural of "automaton"
- Simple (very limited) computational model:
  - Not a programming language or a technology
  - No user-interface, no database back-end
- FSAs check if a string of characters adhere to syntax
  - Variables in programming languages: letters, numbers and "_"
  - Numbers in programming languages: numbers, "e", ".", "+", "−"
- What can you do with FSAs? Plenty!
  - Checking if variables, numbers, etc. are in conformance with syntax
  - Many controller systems (e.g., thermostat) use FSAs

Preliminaries II

Terminology and convention

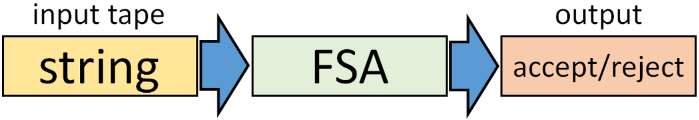
- **Symbols**: atomic (basic) components of a formal language
  - Examples: digits 0–9, small letters a–z, special characters %, ∗, etc.
  - Symbols cannot be split apart – they are like keys of a keyboard
- **Alphabet**: a **finite** set of symbols
  - Referred to as Σ (capital sigma)
  - Examples: $\Sigma_1 = \{0, 1, \ldots, 9\}$, $\Sigma_2 = \{a, b, \ldots, z\}$
- **String** over alphabet Σ: a finite sequence of symbols from Σ
  - Example: strings over $\Sigma = \{0, 1\}$ are $0, 00, 010$, and so on
  - A special **empty string** ϵ (epsilon) with no symbols
  - Some textbooks represent the empty string as λ (lambda)

Intuition

We define a computational model consisting of

- An input tape where there is a string to be checked
- An FSA
- The output ("accept" or "reject")

Graphically:

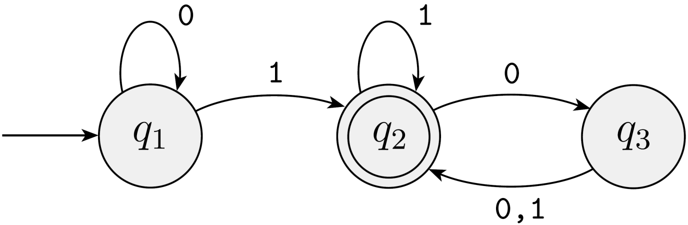


Assumptions:

- Tape has a clear "end-of-string" marker
- FSA handles any input characters (no "exceptions")

Transition graph

- FSAs formally defined in terms of sets, functions, etc.
- Graphic representation: **transition graph** (better for humans)



- Each node/vertex (circle) is a **state**
  - Initial state: q_1 (indicated by unlabelled arrow)
  - Final/accepting state: q_2 (indicated by double circle)
- States connected via **labelled transitions** (edges)
- Labels of transitions compared against characters from input tape

How FSAs Work I

- A “head” (indicated with “$\Downarrow$”) reads current tape cell
- At the beginning head is “ready to go”, to the left of leftmost cell

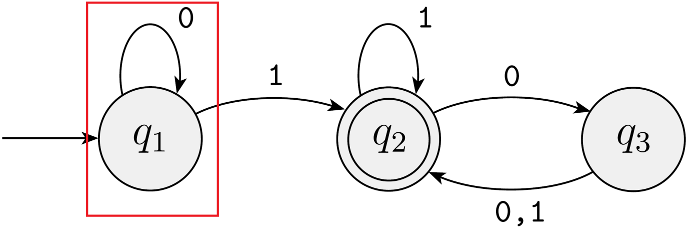
$\Downarrow$

1	0	0	1	$\diamond$

  - “$\diamond$” clearly marks end of string

How FSAs Work II

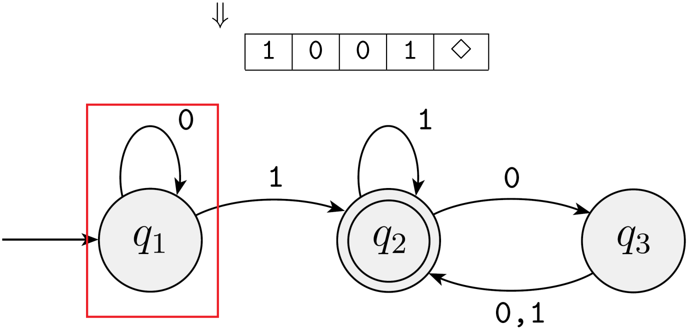
- FSA has a "current state" (indicated via a red square)
  - At the beginning current state is initial state q_1



We'll show next just the input tape and the FSA working together

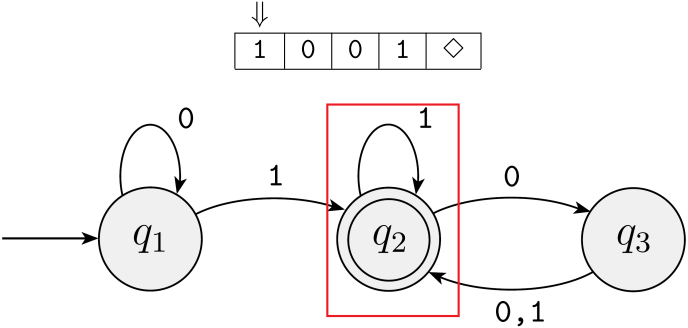
How FSAs Work III

Step 1:



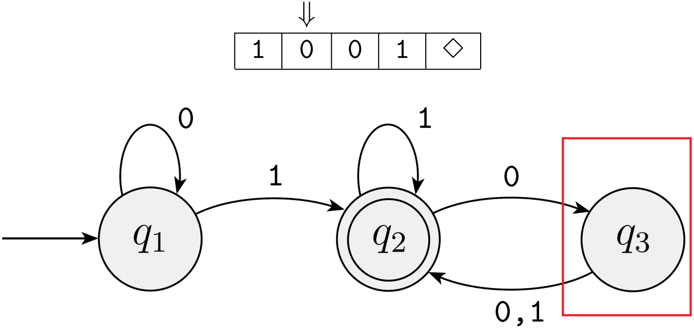
How FSAs Work IV

Step 2:



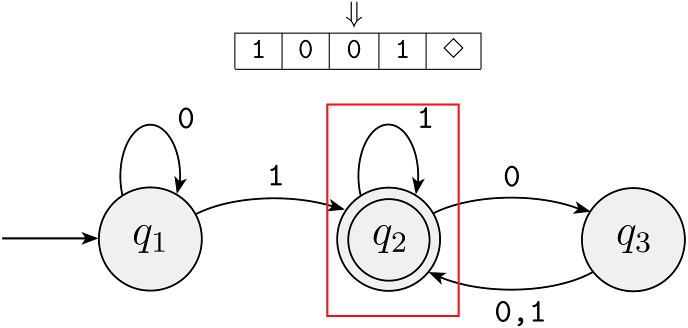
How FSAs Work V

Step 3:



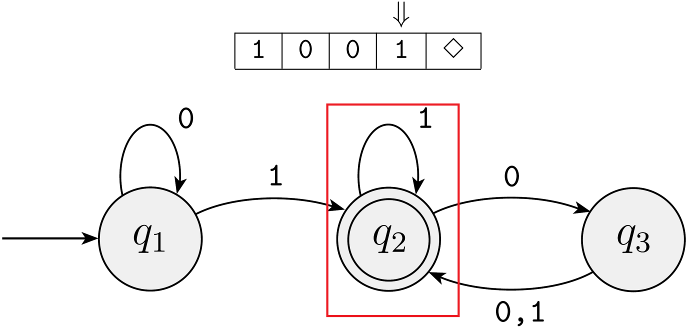
How FSAs Work VI

Step 4:



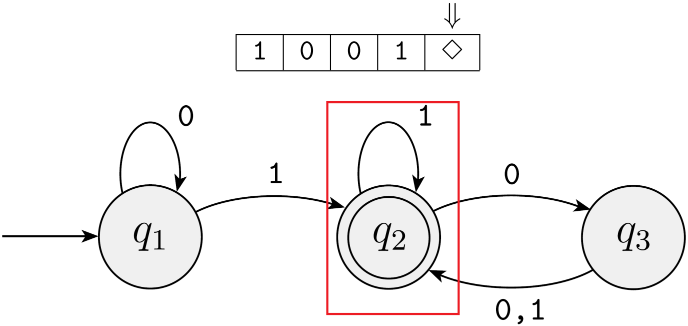
How FSAs Work VII

Step 5:



How FSAs Work VIII

Step 6:



- We reached the end of the input string
- If current state is a final state, then **accept**
- If current state not a final state, then **reject**

Formal Definition of FSA I

- Graphic representation is useful for humans
- However, we need **precision** and **compactness**
  - Must FSAs have exactly one accepting state?
  - Must FSAs have a transition for all symbols in each state?
- We define FSAs formally via sets and functions

Definition (Finite State Automaton M)

A finite state automaton M is a tuple $M = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of **states**
- Σ is a finite set of symbols, our **alphabet**
- $\delta : Q \times \Sigma \mapsto Q$ is the **transition function**
- $q_0 \in Q$ is the **initial/start state**
- $F \subseteq Q$ is the set of **accept states**

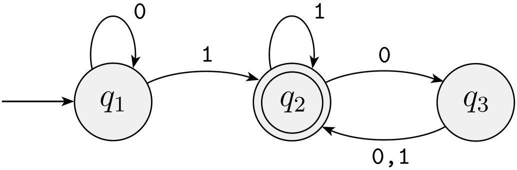
Formal Definition of FSA II

Back to our questions:

- Must FSAs have exactly one accepting state? No, because
  - F can be empty (no accepting states).
  - F can have more than one state.
- Must FSAs have a transition for all symbols in each state? Yes, because
  - δ is a function defined for all values of Q and Σ

Formal Definition of FSA III

Let's formally describe our automaton



Let's call it $M_1 = (Q, \Sigma, \delta, q_1, F)$ where

- $Q = \{q_1, q_2, q_3\}$ and $\Sigma = \{0, 1\}$
- $\delta : Q \times \Sigma$, that is, $\delta : \{q_1, q_2, q_3\} \times \{0, 1\}$

$$\begin{array}{c} \delta(q_1, 0) = q_1,\ \delta(q_1, 1) = q_2, \\ \delta(q_2, 0) = q_3,\ \delta(q_2, 1) = q_2, \\ \delta(q_3, 0) = q_2,\ \delta(q_3, 1) = q_2 \end{array}$$

or as a table:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

- $q_0 = q_1$ (q_1 is our start state)
- $F = \{q_2\}$

Formal Definition of FSA IV

- If A is the set of strings M accepts then
  - We say that A is the **language of machine** M
  - We write $L(M) = A$
  - We say that M **recognises** A of that M **accepts** A
- A machine may accept several (infinite!) strings
- A machine recognises only one language
- If M accepts no strings it still recognises the empty language $L(M) = \emptyset$

How Knowledge on FSAs is Gauged

- Now that we have a formal definition of FSAs, what do we do with it?
- Given an FSA M you should be able to
  - Check whether or not it accepts a particular string
  - Precisely describe what language it accepts
- Given a language description, you should be able to
  - Design an FSA which accepts it
  - Prove that there is not an FSA which accepts it

Formal Definition of Computation

- We introduced an informal model of computation with FSAs
- We need to make it more precise (to avoid ambiguities)

Definition (M accepts w)

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FSA and $w = a_1 a_2 \cdots a_n$ be a string, where each $a_i \in \Sigma$. **M accepts w** if, and only if, there is a sequence of states r_0, $r_1, \ldots, r_n$, $r_j \in Q$, such that the following 3 conditions hold

1. $r_0 = q_0$ (the machine starts in the start state)
2. $\delta(r_i, a_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$(M goes from state to state according to the transition function), and
3. $r_n \in F$ (it accepts the input if it ends up in an accept state)

Definition (Regular languages)

A language is called **regular** if some FSA recognises it.

Designing Finite Automata I

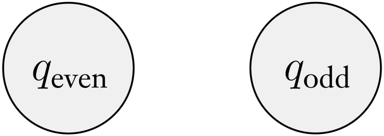
- No simple recipe to design automata
- Suggestion: pretend to be the machine you are designing
  - A psychological trick to help engage your mind in the design process
- You receive an input string and must determine whether it should be accepted
- You get to see the symbols in the string one by one
- After each symbol, you must decide whether the string seen so far is in the language
- You, like the machine, don't know when the end of the string is coming, so you must always be ready with the answer

Designing Finite Automata II

- You need to figure out what you need to "remember" about the string
  - Finite states = finite memory
  - Strings can be arbitrarily long; your FSA should work for any string
- Suppose a language over $\{0, 1\}$ of all strings with an odd number of 1s: $\{1, 01, 10, 1101, \ldots\}$
  - No need to remember all characters to determine if number of 1s is odd
  - Simply remember if number of 1s so far is even/odd
  - Keep track of this information as you read new symbols
  - If you read a 1, flip the answer; if you read a 0, leave answer as is

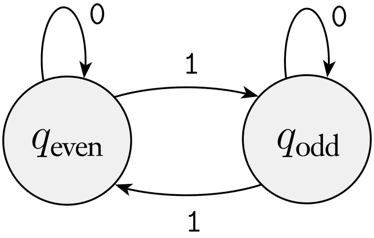
Designing Finite Automata III

- Once you have determined the necessary information to remember about the string as it is being read,
  - Represent this information as a finite list of possibilities
- In our case, the possibilities would be
  1. even so far, and
  2. odd so far
- Then you assign a state to each of the possibilities:



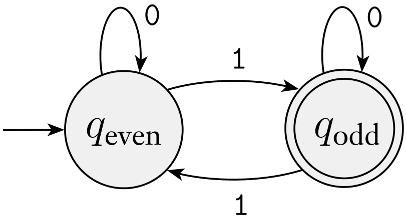
Designing Finite Automata IV

- Next assign the transitions by seeing how to go from one possibility to another upon reading a symbol:
  - q_{even} represents the even possibility
  - q_{odd} represents the odd possibility
- Set transitions to flip state on a 1 and stay put on a 0:



Designing Finite Automata V

- Next set start state to be the state corresponding to the possibility associated with having seen 0 symbols so far (the empty string ϵ)
  - Start state corresponds to q_{even} because 0 is an even number
- Last, set accept states to be those corresponding to possibilities where you want to accept the input string
  - Set q_{odd} to be an accept state because you want to accept when you have seen an odd number of 1s



Final Remarks

- This lecture is not a "crash course" on FSAs
  - It is just to remind ourselves of the topic and introduce formalism
  - It's been a long time...
  - We are aware how you all enjoyed CS2013 ☺
- Assumption: CS2013 material "fresh" in your mind
  - New slides for this year's CS2013
  - Available in our MyAberdeen area for CS3518
- Material from Sipser's book (recommended reading), Chapter 1
- Tutorial next week covering this topic