Automata and Formal Languages (ECS421)

Nikos Tzevelekos

Lecture 1

Introduction

Organisation

- Lecturer: Nikos Tzevelekos
- Team of Demonstrators
- Schedule:
  - Lecture and Exercises: Wednesdays 11:00 – 13:00, weeks 1 – 11
  - Labs: Wednesdays 9:00 – 11:00, weeks 2 – 12
    - *check your assigned slot!*
- Communication:
  - *ask questions during the lectures/labs!*
  - reach me online, either by email or the QM+ forum
- Assessment:
  - 10 lab sheets (10%), 2 courseworks (20%), exam (70%)

More organisation

- Material will be available on QM+ page each week
  - lecture slides, exercises and lab sheets (with solutions)
  - Lecture recordings
- Past papers will be made available on QM+ as well
  - these are good preparation for the exam, along with the exercises we solve in class and lab sheets
  - no expectation that the exam is going to be the same this year, so not a good strategy to memorise past papers
- Coursework assignments will be made available on QM+ 2 weeks before submission
  - *note these are individual assignments, you should not work together on them*

Some Keywords

Concepts:

- Abstract Machines
- Formal Languages and Grammars

Fundamental in CS (algorithms, verification, text-mining, etc.)

Applications:

- Programming, Communicating with machines
- Compilers, Parsing
- Hardware design and verification
- Automata and Formal Languages also used in Linguistics, Engineering, Biology, etc.

Outline

- Week 1: Intro, Chomsky Hierarchy, Finite-State Automata (FSAs)
- Weeks 2-3: FSAs and Regular Expressions
- Week 4: Pumping Lemma, transformations of FSAs
- Week 5: Context-Free Grammars (CFGs)
- Week 6: Pushdown Automata (PDAs)
- Weeks 8-9: Connections between CFGs, PDAs, FSAs
- Weeks 10-11: Parsing
- Weeks 7,12: Revisions (important!)

Books

- Michael Sipser
  Introduction to the Theory of Computation

- John C. Martin
  Introduction to Languages and The Theory of Computation

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman
  Automata Theory, Languages, and Computation

- for parsing:

  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey Ullman
  Compilers: Principles, Techniques, and Tools

Some fundamental terminology

A ***set*** is a collection of objects. Each object is called an *element*.

Example sets:

$\{a,\ b,\ c\}$	the set with elements a, b and c
$\{\ \}$ (also: $\emptyset$)	the empty set (with no elements)
$\{0,\ 1,\ 2,\ \ldots\ \}$ (also: $\mathbb{N}$ or ω)	the set of natural numbers
$\{2x \mid\ x \in \mathbb{N}\ \}$	the set of even natural numbers

Set operators:

$a \in \{a, b\}$ but $c \notin \{a, b\}$	an element belongs to a set (or not)
$\{a, b\} \subseteq \{a, b, c\}$ but $\{a, b, d\} \nsubseteq \{a, b, c\}$	a set is a subset of another set (or not)
$\{a, b\} \cap \{a, c\} = \{a\}$	set intersection
$\{a, b\} \cup \{a, c\} = \{a, b, c\}$	set union

Some fundamental terminology

A ***pair*** of two elements is denoted as: $(a,\ b),\ (b,\ 0),\ \ldots$

A ***tuple*** is a pair that can have more than 2 components: $(a, b, q, 0)$

- note that tuples are *ordered*, while sets are not!

  For example: $(\mathrm{a,b,c}) \neq \{\mathrm{a,b,c}\}$

A ***string***, or ***word***, is a sequence of elements of arbitrary length (even zero): $a,\ ab,\ baaab,\ aaabaab,\ \ldots,\ \varepsilon$

$\boxed{\varepsilon \text{ = the } \textit{empty} \text{ word}}$

Set operators (continued):

$\{a,\ b\} \times \{0,\ 1,\ 2\} = \{(a,\ 0),\ (a,\ 1),\ (a,\ 2),\ (b,\ 0),\ (b,\ 1),\ (b,\ 2)\}$	set product
$\{a,\ b\}^* = \{\varepsilon,\ a,\ b,\ aa,\ bb,\ ab,\ ba,\ aaa,\ bbb,\ baa,\ bba,\ \ldots\}$	Kleene star

Languages and Computation

A ***language*** is a set of strings over a specified (finite) set of input symbols, which is called an ***alphabet***.

- I.e. a language over an alphabet Σ is a subset of Σ^*

In this module we will study different ways of *expressing* (or *recognising*) languages:

- via abstract machines
- via grammars

Why this matters:

- Computation = acceptance of specified languages
  - e.g. computing a function f is the same as accepting $\{(x, f(x)) \mid x \in \mathbb{N}\}$
- To program computers, implement machine communication, etc. one needs to use formal languages and protocols

Example languages

Here are some example languages (and their alphabets Σ):

- Given $\Sigma = \{a, b\}$, some simple languages are:

$$\{a\} \qquad \{b\} \qquad \{a, b\} \qquad \{ab,\ ba\}$$

- Given $\Sigma = \{a, b\}$, some less simple languages are:
  - $L_{\text{all-}a} = \{\varepsilon,\ a,\ aa,\ aa\ ,\ aaa,\ aaaa,\ aaaaa,\ \ldots\ \}$
  - $L_{\text{even-}a} = \{\varepsilon,\ aa,\ aaaa,\ aaaaaa,\ \ldots\ \}$
  - $L'_{\text{even-}a} = \{\ w \in \Sigma^* \mid w$ contains an even number of a's $\}$
- Given any Σ, two (different!) special languages are:

$$L_{\phi} = \emptyset \qquad L_{\varepsilon} = \{\varepsilon\}$$

how are they different?

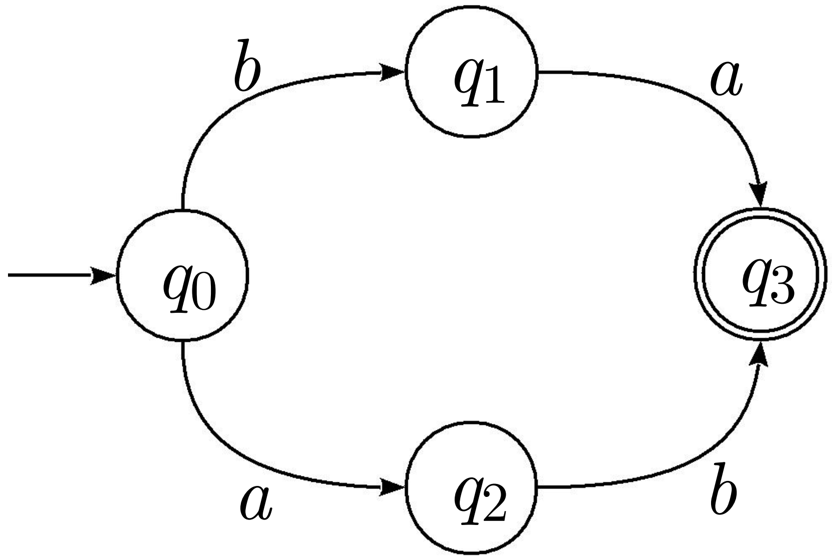
Finite-state automata

A ***finite-state automaton (FSA)*** is a graph where nodes are *states* and arrows are *transitions* :

- there is a unique *initial* state, and several *final* states
- transitions are labelled with letters from the alphabet Σ

E.g. here is an FSA A :

- $\Sigma = \{a, b\}$
- $L(A) = \{ab, ba\}$



Computation/acceptance of a word is done by following transitions from the initial state to any final state

The coffee machine FSA

The coffee company asks us to design a coffee machine that dispenses coffee and tea. The specs are:

- The machine only accepts 50p coins
- Coffee costs £1, double coffee is £1.50, tea is £0.50
- First the correct amount is inserted, then the beverage button is pressed
- The machine breaks if the specs are not followed (e.g. do not insert 20p coins!)

We design an FSA that:

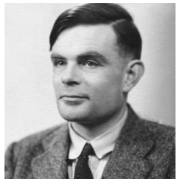
- has 4 states, one initial/final and one for each 50p inserted
- has alphabet $\Sigma = \{$50p, CF, 2CF, TEA$\}$

Then extend the specs to accept £1 coins

More languages: the Chomsky Hierarchy

Different languages may require more/less involved machines (or grammars) for their description.

Classification of formal languages by Noam Chomsky:

- Type-0: Recursively enumerable languages, all formal grammars, Turing machines (not  but )
- Type-1: Context-sensitive languages/grammars, linear-bounded Turing machines
- **Type-2**: Context-free languages/grammars, pushdown automata
- **Type-3**: Regular languages/grammars, finite-state automata

Properties of the Chomsky Hierarchy

Universality : Type-0 languages represent all “computable” languages/algorithms/programs

Inclusion : all Type-3 languages are Type-2, all Type-2 are Type-1, all Type-1 are Type-0

Strictness : there exist languages that are Type-0 but not Type-1, Type-1 but not Type-2, Type-2 but not Type-3

Turing Machines

Based on the idea of a “computor” (a human who computes):

- finitely many control states, but
- a potentially infinite “tape” to write/read results

This is Alan Turing's (equivalent) description of what we accept as computable functions (*Church-Turing thesis*)

- any algorithm/program can be broken down to a Turing machine computation via appropriate encodings
- Turing machines are the very basis of abstract machines
- Type-0 languages: recognisable by TMs
- Type-1: recognisable by TMs that cannot write/read beyond the input part of their tape (linearly bounded)

more on TMs in ECS651U!

Context-free and regular languages

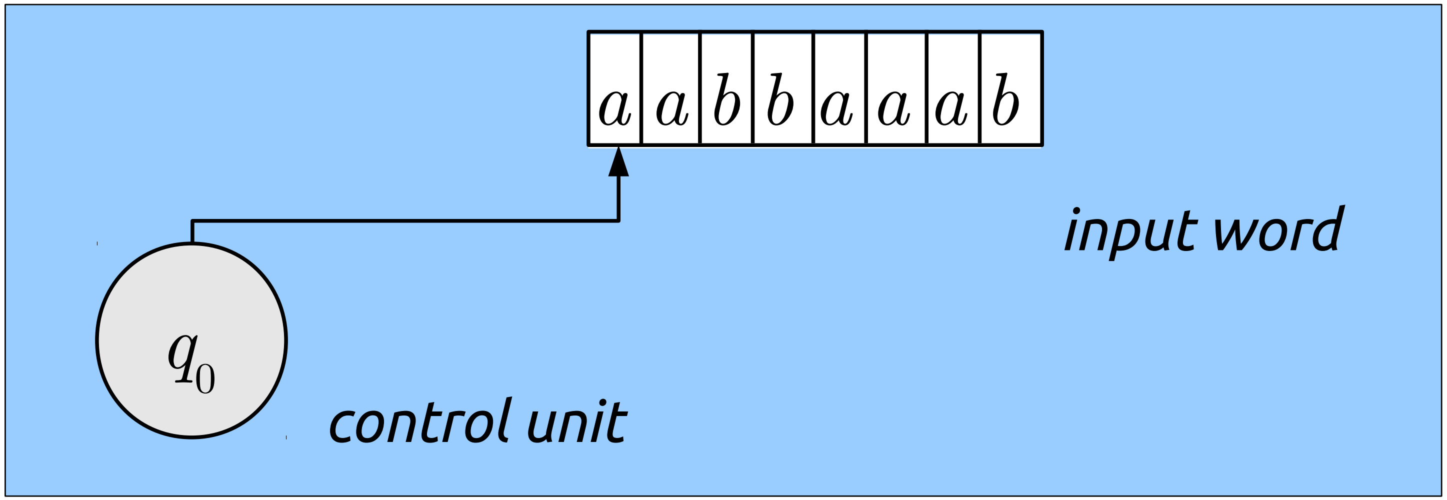
In this module we will study Type-2 and Type-3 languages

- Type-2 languages are called ***context-free***.

  They correspond to *pushdown automata*: TMs where the tape is broken into a *read-once input* and a *stack*

- Type-3 languages are called ***regular***.

  They correspond to finite-state automata: TMs where there is only a read-once input and finite control



Quiz time (get ready!)

`go to http:://kahoot.it`

Quiz time (get ready!)

Formal languages are:
- protocol languages
- programming languages
- sets of words over a given alphabet
- all of the above

A Turing machine is:
- one of the first computers ever constructed
- a programming language
- an abstract device capturing computation processes
- a device for breaking WWII encryption

The Chomsky hierarchy categorises formal languages by:
- the class of abstract machines that recognise them
- the class of grammars that recognise them
- both

Finite-state automata can express:
- the same languages as Turing machines
- more languages than Turing machines
- fewer languages than Turing machines

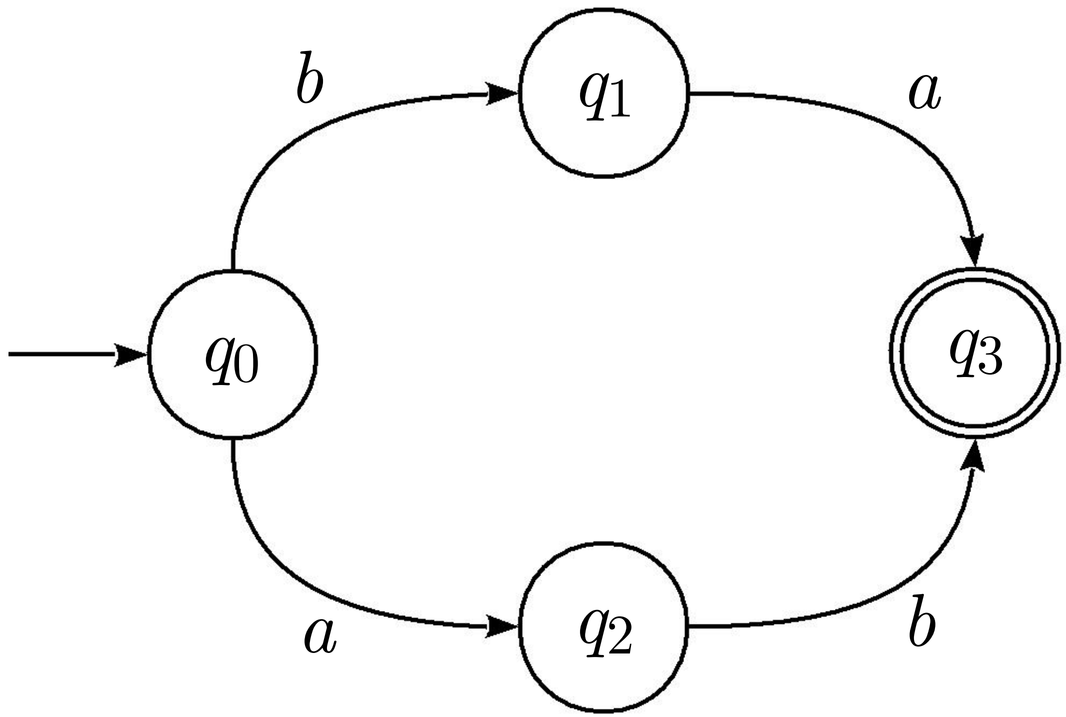
Formal definition

A ***Finite-State Automaton (FSA)*** is a 5-tuple

$$A = (\Sigma,\ Q,\ \delta,\ q_0,\ F)$$

- Σ is the *input alphabet*
- Q is the finite set of *control states*
- δ is the *transition relation*, with $\delta \subseteq Q \times \Sigma \times Q$
- q_0 is the *initial state*, with $q_0 \in Q$
- F is the set of *final (or accepting) states*, with $F \subseteq Q$

Formal definition vs graph notation



The above FSA is given by:

$$A = (\{a, b\}, \{q_0, q_1, q_2, q_3\}, \{(q_0, b, q_1), (q_0, a, q_2), (q_1, a, q_3), (q_2, b, q_3)\}, q_0, \{q_3\})$$

Summary

In this introductory lecture we:

- saw how great ECS421 is! :)
- talked about Formal Languages and Abstract Machines
- looked at the Chomsky Hierarchy
- defined Finite-State Automata and examined examples

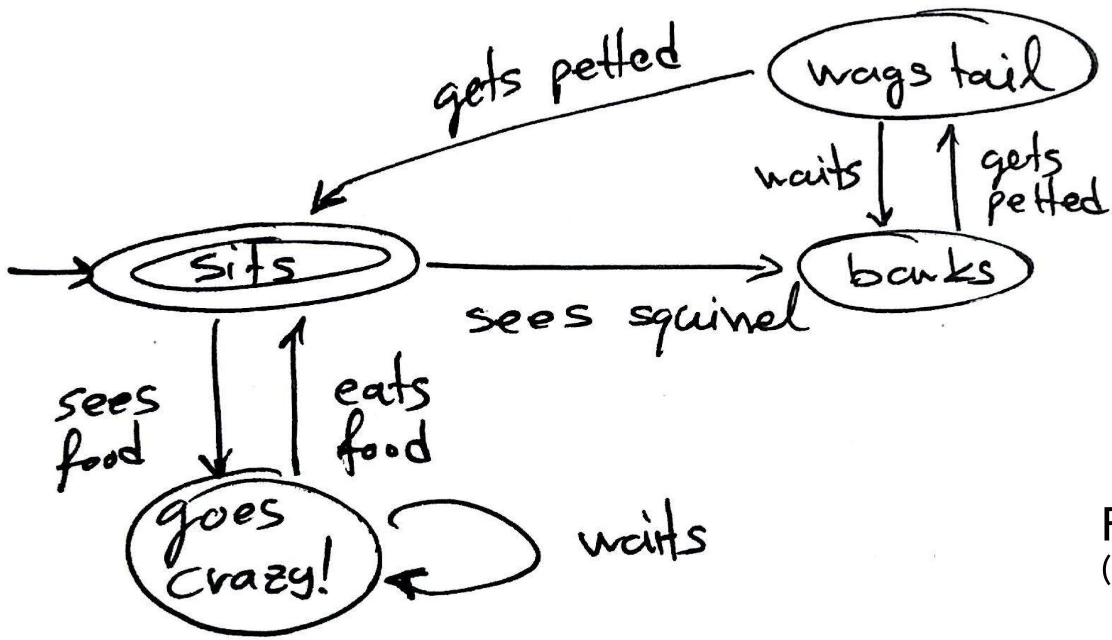


Fig 1. My dog as an FSA
(idea from learnyousomeerlang.com)