Finite Automata Theory and Formal Languages

TMV027/DIT321– LP4 2017

Lecture 1
Ana Bove

March 20th 2017

Overview of today's lecture:

- Overview of the course;
- Course organisation.

Automaton

Dictionary definition:

```
Main Entry: au·tom·a·ton
Function: noun
Inflected Form(s): plural au·tom·atons or au·tom·a·ta
Etymology: Latin, from Greek, neuter of automatos
Date: 1645

1 : a mechanism that is relatively self-operating;
    especially : robot
2 : a machine or control mechanism designed to follow
    automatically a predetermined sequence of operations or
    respond to encoded instructions
3 : an individual who acts in a mechanical fashion
```

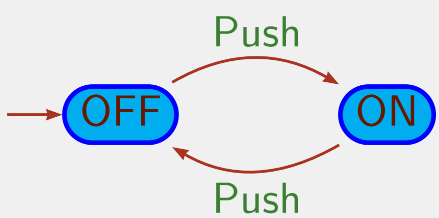
Automata: Applications

Models for ...

- Lexical analyser in a compiler;
- Software for designing circuits;
- Software for finding patterns in large bodies of text such as collection of web pages;
- Software for verifying systems with a finite number of different states such as protocols;
- Real machines like vending machines, telephones, street lights, ...;
- Application in linguistic, building of large dictionary, spell programs, search;
- Application in genetics, regular pattern in the language of protein.

Example: on/off-switch

A very simple finite automaton:



States represented by “circles”.

<u>One</u> *starting* state, indicated with an arrow into it.

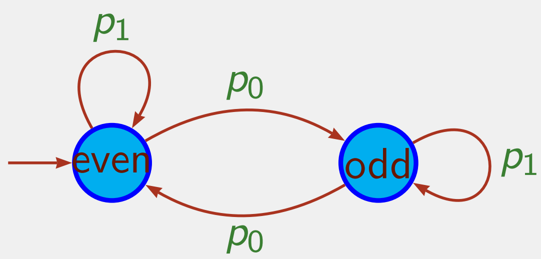
Labelled arcs between states represent observable *events*.

Sometimes <u>one or more</u> *final* states, indicated with a double circle.

Example: Parity Counter

The states of an automaton can be thought of as its *memory*.

A finite-state automaton has *finite memory*!



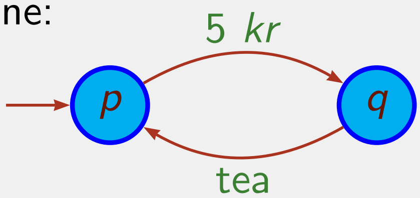
Two events: p_0 and p_1.

The machine does nothing on the event p_1.
The machine remembers the parity of the number of p_0's.

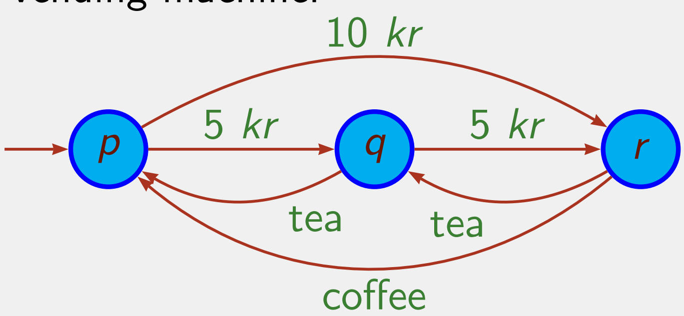
Correctness: We could prove that the automata is on the state even iff an even number of p_0 were pressed.

Example: Vending Machines

A simple vending machine:



A more complex vending machine:



What does it happen if we ask for a tea on p?

Example: The Man, the Wolf, the Goat and the Cabbage

A man with a wolf, a goat and a cabbage is on the left bank of a river.

There is a boat large enough to carry the man and only one of the other three things. The man wish to cross everything to the right bank.

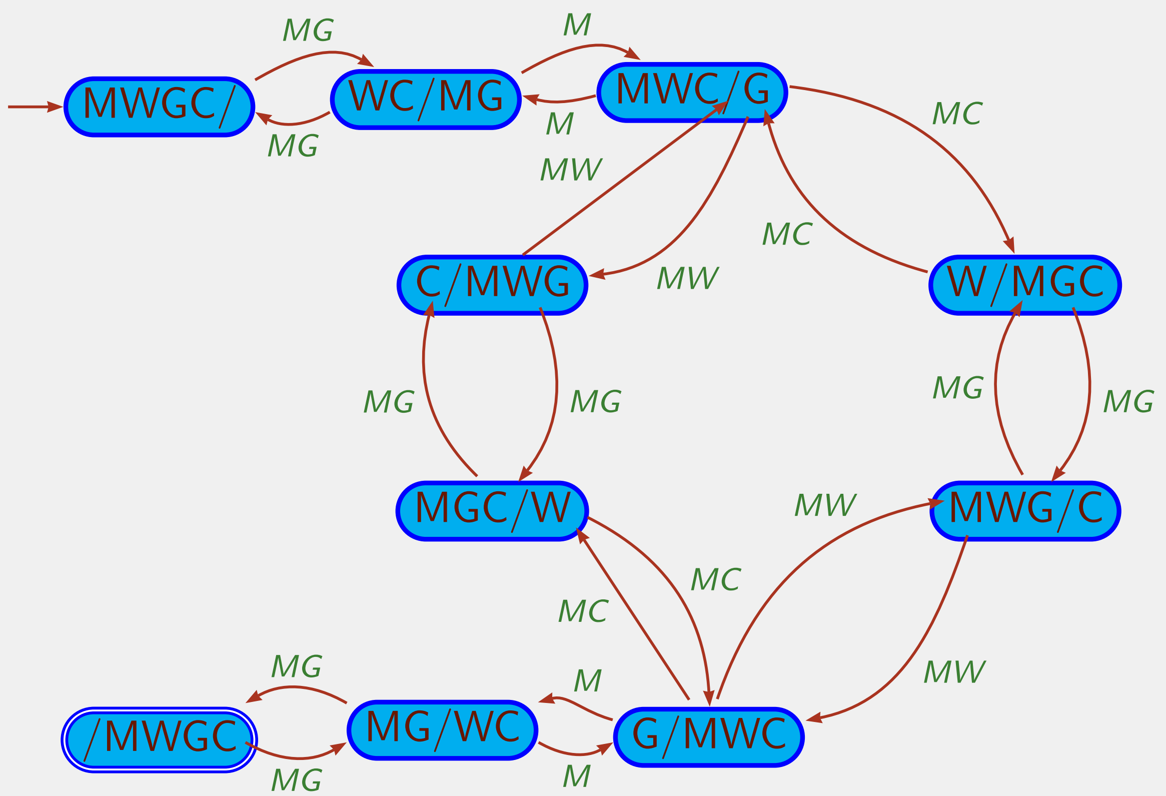
However if the man leaves the wolf and the goat unattended on either shore, the wolf surely will eat the goat.

Similarly, if the goat and the cabbage are left unattended, the goat will eat the cabbage.

Problem: Is it possible to cross the river without the goat or cabbage being eaten?
How many possible solutions the problem has?

Solution: We design an automaton that models the problem with all its possible transitions, and look for paths between the initial and final state.

Solution: The Man, the Wolf, the Goat and the Cabbage



Formal Languages

From Wikipedia:

```
In mathematics, computer science, and linguistics, a formal
language is a set of strings of symbols that may be
constrained by rules that are specific to it.

The alphabet of a formal language is the set of symbols,
letters, or tokens from which the strings of the language
may be formed; frequently it is required to be finite.

The strings formed from this alphabet are called words, and
the words that belong to a particular formal language are
sometimes called well-formed words or well-formed formulas.

A formal language is often defined by means of a formal
grammar such as a regular grammar or context-free grammar,
also called its formation rule.
```


Example: Formal Representation of Numbers and Identifiers in a Programming Language

A *regular grammar* for numbers and identifiers

$$\begin{array}{lcl} L & \rightarrow & \mathbf{A} \mid \mathbf{B} \mid \ldots \mid \mathbf{Z} \mid \mathbf{a} \mid \mathbf{b} \mid \ldots \mid \mathbf{z} \\ D & \rightarrow & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \\ \\ Nr & \rightarrow & D \mid D\ Nr \\ Id & \rightarrow & L\ LLoD \\ LLoD & \rightarrow & L\ LLoD \mid D\ LLoD \mid \epsilon \end{array}$$

A *regular expression* for numbers:

$$(\mathbf{0} + \mathbf{1} + \mathbf{2} + \mathbf{3} + \mathbf{4} + \mathbf{5} + \mathbf{6} + \mathbf{7} + \mathbf{8} + \mathbf{9})^+$$

and for identifiers:

$$(\mathbf{A} + \ldots + \mathbf{Z} + \mathbf{a} + \ldots + \mathbf{z})(\mathbf{A} + \ldots + \mathbf{Z} + \mathbf{a} + \ldots + \mathbf{z} + \mathbf{0} + \ldots + \mathbf{9})^*$$

Example: Very Simple Expressions

A *context-free grammar* for simple expression:

$$\begin{array}{lcl} E & \rightarrow & E{+}E \mid E{-}E \mid E{*}E \mid E/E \mid (E) \mid Nr \mid Id \\ Nr & \rightarrow & \cdots \\ Id & \rightarrow & \cdots \end{array}$$

Correctness: We could prove that

- Any expression has as many "(" as ")";
- Any expression has 1 more "term" than the number of "operations";
- ...

More Complex Examples

An equivalent (but better) context-free grammar for simple expression:

$$\begin{array}{lcl} E & \rightarrow & E{+}T \mid E{-}T \mid T \\ T & \rightarrow & T{*}F \mid T/F \mid F \\ F & \rightarrow & (E) \mid Nr \mid Id \end{array}$$

A context-free grammar for C++ compound statements:

$$\begin{array}{lcl} S & \rightarrow & \{LC\} \\ LC & \rightarrow & \epsilon \mid C\ LC \\ C & \rightarrow & S \mid \textbf{if}\ (E)\ C \mid \textbf{if}\ (E)\ C\ \textbf{else}\ C \mid \\ & & \textbf{while}\ (E)\ C \mid \textbf{do}\ C\ \textbf{while}\ (E) \mid \textbf{for}\ (C\ E;E)\ C \mid \\ & & \textbf{case}\ E{:}C \mid \textbf{switch}\ (E)\ C \mid \textbf{return}\ E; \mid \textbf{goto}\ Id; \\ & & \textbf{break}; \mid \textbf{continue}; \\ & \vdots & \end{array}$$

Overview of the Course

- Formal proofs;
- Regular languages;
- Context-free languages;
- Turing machines (as much as time allows).

Formal Proofs

Many times you will need to prove that your program/model/grammar/... is "correct" (satisfies a certain specification/property).

In particular, you won't get a complex program/model/grammar/... right if you don't understand what is going on.

Different kind of formal proofs:

- Deductive proofs;
- Proofs by contradiction;
- Proofs by counterexamples;
- Proofs by (structural) induction.

Regular Languages

Finite automata were originally proposed in the 1940's as models of neural networks.

Turned out to have many other applications!

In the 1950s, the mathematician Stephen Kleene described these models using mathematical notation (*regular expressions*, 1956).

Ken Thompson used the notion of regular expressions introduced by Kleene in the UNIX system.

(Observe that Kleene's regular expressions are not really the same as UNIX's regular expressions.)

Both formalisms define the *regular languages*.

Context-Free Languages

We can give a bit more power to finite automata by adding a stack that contains data and obtain a *push down automata*.

In the mid-1950s Noam Chomsky developed the *context-free grammars*.

Context-free grammars play a central role in the description and design of programming languages and compilers.

Both formalisms define the *context-free languages*.

Turing Machine (ca 1936–7)

Simple theoretical device that manipulates symbols contained on a tape.

It is as "powerful" as the computers we know today (in terms of what they can compute).

It allows the study of *decidability*: what can or cannot be done by a computer (*halting* problem).

Computability vs *complexity* theory: we should distinguish between what can or cannot be done by a computer, and the inherent difficulty of the problem (*tractable* (polynomial)/*intractable* (NP-hard) problems).

Church-Turing Thesis

In the 1930's there has been quite a lot of work about the nature of *effectively computable (calculable) functions*:

- Recursive functions by Stephen Kleene (after ideas by Kurt Gödel);
- λ-calculus by Alonzo Church;
- Turing machines by Alan Turing.

The three *models of computation* were shown to be equivalent by Church, Kleene & (John Barkley) Rosser (1934–6) and Turing (1936-7).

The *Church-Turing thesis* states that if an algorithm (a procedure that terminates) exists then, there is a Turing machine, a recursively-definable function, or a definable λ-function for that algorithm.

Learning Outcome of the Course

After completion of this course, the student should be able to:

- Explain and manipulate the different concepts in automata theory and formal languages;
- Have a clear understanding about the equivalence between (non-)deterministic finite automata and regular expressions;
- Understand the power and the limitations of regular languages and context-free languages;
- Prove properties of languages, grammars and automata with rigorously formal mathematical methods;
- Design automata, regular expressions and context-free grammars accepting or generating a certain language;
- Describe the language accepted by an automata or generated by a regular expression or a context-free grammar;
- Simplify automata and context-free grammars;
- Determine if a certain word belongs to a language;
- Define Turing machines performing simple tasks;
- Differentiate and manipulate formal descriptions of languages, automata and grammars.

People and Contact Information

Course Responsible and Examiner:
Ana Bove, `bove@chalmers.se`

Assistants:
Victor López, `lopezv@chalmers.se`
Daniel Schoepe, `schoepe@chalmers.se`
Marco Vassena, `vassena@chalmers.se`
Andrea Vezzosi, `vezzosi@chalmers.se`

Course Mailing List: `fafl@lists.chalmers.se`

Important information related to the course will be sent there.

You will get registered to it next week with your CTH/GU mail address.
The list is moderated.

Course Level, Load and Web Page

Level: This course is a *bachelor* course in year 1–2.

Load: 7.5 pts means ca. *20–25 hours per week*!!

Note: It is important to follow the course's pace in order to pass it!

Web Page: `http://www.cse.chalmers.se/edu/course/TMV027`

Accessible from CTH "studieportalen" and GU "GUL".
Check it regularly for news!

Literature and Other Material

Main book: *Introduction to Automata Theory, Languages, and Computation*, by Hopcroft, Motwani and Ullman. Addison-Wesley.

We will cover chapters 1 to 5, 7 and a bit of chapter 8.

Note: Notation in the book is sometimes different to that used in class.

Online book: *Mathematics for Computer Science*, by Lehman, Leighton and Meyer

For recap on discrete math concepts and the part on formal proofs.

Wikipedia: `http://en.wikipedia.org/wiki`

Youtube: `http://www.youtube.com`

Course Organisation

The course has an introductory part, 7 "modules" and a closing part.

Each module consists of:

- Two lectures by Ana;
- Two exercises classes with the same content by the assistants;
- A consultation time by Ana: in group we discuss whatever you need help with;
- An individual help session by assistants: when you get stuck with a particular exercise;
- An *individual assignment*.

Note: Check time edit/course web page for details!

Note: You *MUST prepare* before each lecture in order to follow the course pace!

Solutions to the Exercises

Note: There are NO solutions to exercises!!!

Why?

- Expensive to create and maintain good solutions;
- Some problems have many possible solutions;
- Pedagogically not always good: you learn mostly by doing;
- Asking the teachers in case of doubt is a better way to learn;
- You need to learn how to solve problems without a solution....
  Who will pay you to solve something one already has the solution to?

However:

- The book's web page have some solutions available;
- There are plenty of other exercises with their solutions on the web!

Online Tools

JFLAP: `http://www.jflap.org`

Can help you finding a right solution to some problems.

Use the tool *wisely*: to learn and not just to copy a solution!

Automata tutor: `http://automatatutor.com`

You can test your knowledge on some exercises here.

You need to create an account and register to the following course:

Course ID: 128FAFL-2
Password: LKO0EKYV

Visit course web page on *Useful links*!

Programming Bits in the Course

The course doesn't require much programming tasks.

Still

- I will present some Haskell programs simulating certain automaton or implementing an algorithm.

  (Code should be easy enough to follow even if you do not know Haskell.)

- You are encourage to implement the algorithms to improve your knowledge and understanding, and to test your solutions!

Examination

From VT2013 the course has 2 *obligatory* parts:

- *Individual weekly assignments:* 1.5pts.
- *Individual written exam:* 6pts, no book or help (but dictionaries) allowed.
  Dates for 2017: May 30th pm and August 16th am.

From VT2015 the final grade is based on the performance on *both* parts!

Visit course web page on *Examination*!

More on Assignments

- To pass the assignment part you need to get at least 50% of the sum of the points of all the weekly assignments together.
- No re-submission possibility.
- No need to submit *all* of them but *VERY* good if you do!
- They must be done *completely on your own*!

  Note: Be aware that assignments are part of the examination of the course and they **should** be done *individually*!
  Standard procedure will be followed if very similar solutions are detected.

Submission of Assignments

How? Via the Fire system, check course web page on Assignments.

Deadline account: Create an account in Fire on *Tuesday 28th March* at the latest!

Who? All students registered or re-registered in the course and who have *NOT* passed them yet!

When? See deadlines on course web page on Assignments.

Course Evaluation

I need 4-5 student representatives from both Chalmers and GU on *Thursday* at the latest.

Otherwise, we will pick randomly.

First meeting on first week of April.

Comments and Advices from Course Evaluation

Tell people even more how much it really is to take in, and how extremely important it is to be in phase.

Go to the exercise classes and the consultation times. It made it all more clear and I actually didn't have to put in much extra work.

I also liked that there weren't any answers to the exercises; though scary at the beginning, it gave a deeper understanding of the subject of the exercise when one had to think twice and trust one's own solution. It was also a great practice for the exam and future employment.

Comments and Advices from Course Evaluation (Cont.)

The assignments were really great and probably the main reason why I liked this course so much. Also, the feedback from the assignments were absolutely great. Really fantastic feedback, very very useful.

The assignments were extremely helpful and most probably the reason on why I passed this course exam.

A high tempo of assignments kept me on my toes and made it clear what I had learned/needed to practice on.

I know the assignements might seem a bit difficult; what's with the whole "no working together, no help and no re-do's" but it's really not that bad. Especailly if you go to the exercises. Seriously, go to the exercises.

Changes from Last Year

Not many changes, quite an stable course by now...

New:

- Individual help sessions;
- Added some time on inductive proofs;
- Automata tutor (send me feedback of you try it!).

Overview of this Week

Mon 20	Tue 21	Wed 22	Thu 23	Fri 24
	Lec 10-12 HB3 Recap logic, sets, relations and functions.		**Ex 10-12 HC3** Recap exercises.	
Lec 13-15 HB3 Overview and organistion.			**Lec 13-15 HB3** Formal proofs. Ind over $\mathbb{N}$. Inductive sets.	**Lec 13-15 HB3** Structural ind. Concepts of automa theory.

Overview of Next Lecture

See chapters 3 and 4 in the *Mathematics for Computer Science* book (or relevant chapters in other text on discrete math).

Recap on:

- Logic;
- Sets;
- Relations;
- Functions.