Finite Automata and Formal Languages

TMV026/DIT321– LP4 2011

Ana Bove

Lecture 1

March 21st 2011

Overview of today's lecture:

- **Course Organisation**
- **Overview of the Course**

Course Organisation

Level: This course is a *bachelor* course.

Lectures: Mondays 13.15–15 in HA2 and Tuesdays 10–11:45 in HC3
Extra lecture on the first week: Thursday 10–11:45 in HA3.

Ana Bove, `bove@chalmers.se`

Exercise Sessions: Thursdays 13.15–15 in EB
VERY important!!
Willard Rafnsson, `willard.rafnsson@chalmers.se`

Exams: May 27th and August 26th.

No books or written help during the exam.

Course Organisation (Cont.)

Assignments: There will be non-obligatory weekly assignments which will generate bonus points valid *ONLY* on the exams during 2011.

Each assignment will be up to 10 pts. If a student has collected n pts in assignments, this will correspond to $n/10$ points in the exam.

Exams are usually 60 points.

Write your *name* and *personal number* when you submit an assignment.

How to submit? On paper or by email. See the web page of the course.

Note: Be aware that assignments are part of the examination of the course. Hence, they should be done *individually*!

Standard procedure will be followed if copied solutions are detected.

Course Organisation (Cont.)

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

Both second or third edition are fine.

We will cover chapters 1 to 7 and a bit of chapter 8 (if times allows).

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

Web Page: `http://www.cse.chalmers.se/edu/course/TMV026/`

Accessible from Chalmers "studieportalen".

Check it regularly!

Course Evaluation: I need 2 student representatives (1 GU + 1 CTH?) by Thursday this week.

See web page of the course for last year's evaluation.

Programming Bits in the Course

The course doesn't require much programming tasks.

Still we will present several algorithms to translate between different formalisations.

I will sometimes show a Haskell program simulating certain automaton or implementing an algorithm.

(Most of you should know Haskell, if not I really recommend you learn it: it is very elegant and nice!)

You are welcome to test your knowledge implementing things in your favourite language!

Automata

Dictionary definition:

```
Main Entry: automaton
Function: noun
Inflected Form(s): plural automatons or automata
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
```

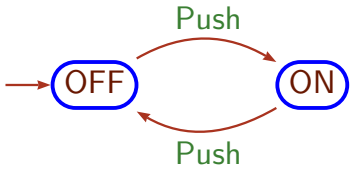
Automata: Applications

Models for ...

- Software for designing circuits
- Lexical analyser in a compiler
- 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 simple non-trivial finite automaton:



States represented by "circles".

One state is the *start* state, indicated with an arrow into it.

Arcs between states are labelled by observable *events*.

Often we need one or more *final* states, indicated with a double circle.

Functional Description of on/off-switch

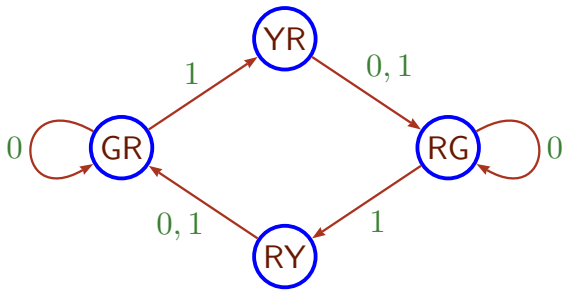
Let us define 2 functions f_{OFF} and f_{ON} representing the 2 states of the automaton.

The input can be represented as a "finite list" give by $N = 0 \mid \mathsf{P}\ N$.

The description of the automaton is: $f_{\mathsf{OFF}}, f_{\mathsf{ON}} : N \to \{\mathsf{Off}, \mathsf{On}\}$

$$f_{\mathsf{OFF}}\ 0 = \mathsf{Off} \qquad f_{\mathsf{ON}}\ 0 = \mathsf{On}$$
$$f_{\mathsf{OFF}}\ (\mathsf{P}\ n) = f_{\mathsf{ON}}\ n \qquad f_{\mathsf{ON}}\ (\mathsf{P}\ n) = f_{\mathsf{OFF}}\ n$$

Example: Simple Traffic Signal

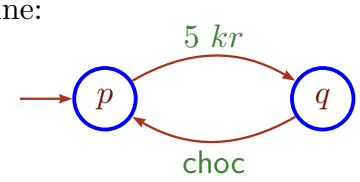


Transitions: 0 (no traffic detected), 1 (traffic detected).

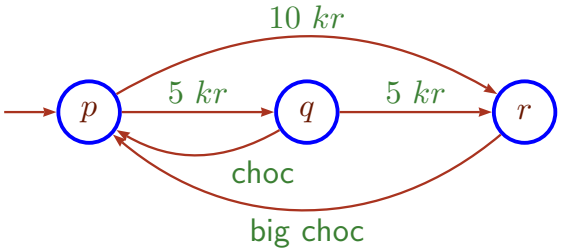
The states represent a situation: GR one traffic light is Green and the other is Red.

Example: Vending Machines

A simple vending machine:



A more complex vending machine:



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

State q remembers it had got 5 kr already.

Problem: 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.

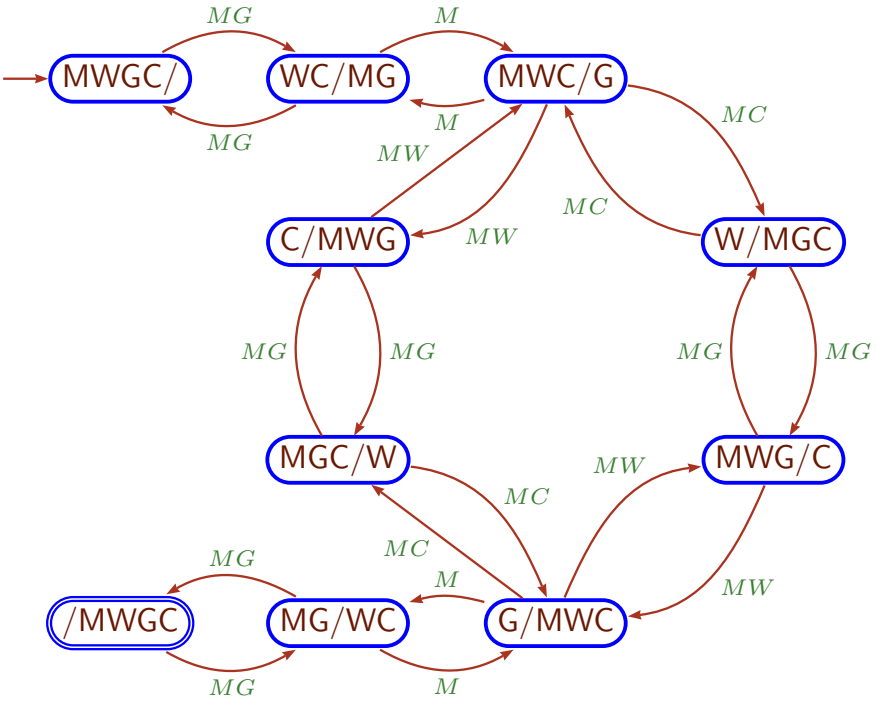
However if the man leaves the wolf and the goat unattended on either shore, the wolf surely will eat the goat.

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

Puzzle: Is it possible to cross the river without the goat or cabbage being eaten?

Solution: We write all the possible transitions, and look for possible paths between two nodes.

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



Overview of the Course

- Formal proofs
- Regular languages
- Context-free languages
- (Turing machines)

Formal Proofs

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

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

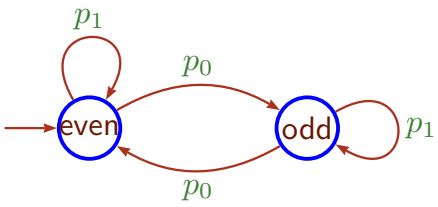
Example: To make an iteration/recursion correct one needs an inductive hypothesis which is consistent with the iteration/recursion.

Different kind of formal proofs:

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

Example: Parity Counter

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



Two events: p_0 and p_1.

The machine does nothing on the event p_1.

The machine counts the parity of the number of p_0's.

A finite-state automaton has *finite memory*!

Functional Description: Parity Counter

Let us define 2 functions f_{even} and f_{odd} representing the 2 states of the automaton.

The input can be represented by the data type $T = 0 \mid p_0\ T \mid p_1\ T$.

The description of the automaton is: $f_{\mathsf{even}}, f_{\mathsf{odd}} : T \to \{\mathsf{Even}, \mathsf{Odd}\}$

$$\begin{array}{ll} f_{\mathsf{even}}\ 0 = \mathsf{Even} & f_{\mathsf{odd}}\ 0 = \mathsf{Odd} \\ f_{\mathsf{even}}\ (p_0\ n) = f_{\mathsf{odd}}\ n & f_{\mathsf{odd}}\ (p_0\ n) = f_{\mathsf{even}}\ n \\ f_{\mathsf{even}}\ (p_1\ n) = f_{\mathsf{even}}\ n & f_{\mathsf{odd}}\ (p_1\ n) = f_{\mathsf{odd}}\ n \end{array}$$

We now would like to prove that $f_{\mathsf{even}}\ n = \mathsf{Even}$ iff n contains an even number of constructors p_0.

Example: on/off-switch

Recall the description of the on/off-switch.

We would like to prove that:

the automaton is in state OFF after n pushes iff n is even

and

the automaton is in state ON after n pushes iff n is odd.

Alternatively, we could prove that:

$f_{\mathsf{OFF}}\ n = \mathsf{Off}$ iff n is even

and

$f_{\mathsf{ON}}\ n = \mathsf{On}$ iff n is odd.

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.

In this way we extend finite automata into a *push down automata*.

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

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

Both formalisms define the *context-free languages*.

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
- λ-calculus by Alonzo Church
- Turing machines by Alan Turing

The three computational processes were shown to be equivalent by Church, Kleene, (John Barkley) Rosser (1934-6) and Alan Turing (1936-7).

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

Turing Machine (ca 1936–7)

Simple theoretical device that manipulates symbols contained on a strip of tape.

It is as "powerful" as the computers we know today (in term 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).

Learning Outcome of the Course

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

- Apply rigorously formal mathematical methods to prove properties of languages, grammars and automata;
- Understand the theory and principle of automata theory;
- Understand, differentiate and manipulate formal descriptions of languages, automata and grammars with focus on regular and context-free languages, finite automata and regular expressions;
- Use and define automata, regular expressions and context-free grammars to solve problems.