

Theoretical Computer Science

Operations on Languages

Lecture 3 - Manuel Mazzara

Operations

- Operations on **sets** apply also to **languages**
 - A language is a **set of strings**
- Operations on languages are
 - Union
 - Intersection
 - Difference
 - Complement
 - Concatenation
 - Power of n
 - Kleene star/closure

Set operations (1)

- $L_1 \cup L_2$
 - Example:
 $L_1 = \{\varepsilon, a, b, c, \mathbf{bc}, \mathbf{ca}\}$
 $L_2 = \{ba, bb, \mathbf{bc}, \mathbf{ca}, cb, cc\}$
 $L_1 \cup L_2 = \{\varepsilon, a, b, c, ba, bb, bc, ca, cb, cc\}$
- $L_1 \cap L_2$
 - Example: $L_1 \cap L_2 = \{bc, ca\}$

Set operations (2)

- $L_1 \setminus L_2$ (or $L_1 - L_2$)
 - Generally used when $L_2 \subseteq L_1$
 - Example:
 $L_1 = \{ba, bb, bc, ca, cb, cc\}$
 $L_2 = \{bc, ca\}$
 $L_1 \setminus L_2 = \{ba, bb, cb, cc\}$
- $L^c = A^* \setminus L$
 - A is the alphabet over which L is defined
 - Example: $L_1^c =$ set of all strings on $\{a,b,c\}^*$ except the strings of length 2 that start with a 'b' or a 'c'

Concatenation

- $L_1 \cdot L_2$ (or L_1L_2) = $\{x \cdot y \mid x \in L_1, y \in L_2\}$

- Remark: ‘ \cdot ’ is not commutative

- $L_1 \cdot L_2 \neq L_2 \cdot L_1$

- Example

$L_1 = \{\varepsilon, a, b, c, bc, ca\}$

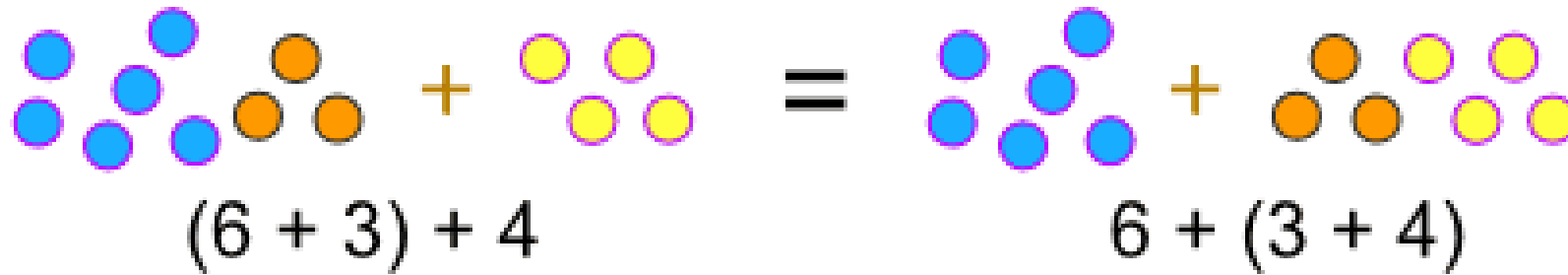
$L_2 = \{ba, bb, bc, ca, cb, cc\}$

$L_1L_2 = \{ba, bb, bc, ca, cb, cc, aba, abb, abc, aca, acb, acc, bba, bbb, bbc, bca, bcb, bcc, cba, cbb, cbc, cca, ccb, ccc, bcba, bcbb, bc bc, bcca, bccb, bccc, caba, cabb, cabc, caca, cacb, cacc\}$

Power

- L^n is obtained by concatenating L with itself n times
 - $L^0 = \{\varepsilon\}$
 - $L^i = L^{i-1} \cdot L$
- Examples:
 - $L^2 = L \cdot L$
 - $L^3 = L \cdot L \cdot L$
 - $L^4 = L \cdot L \cdot L \cdot L$
 - ...
- Remark: ‘ \cdot ’ is associative

Associative Law (1)

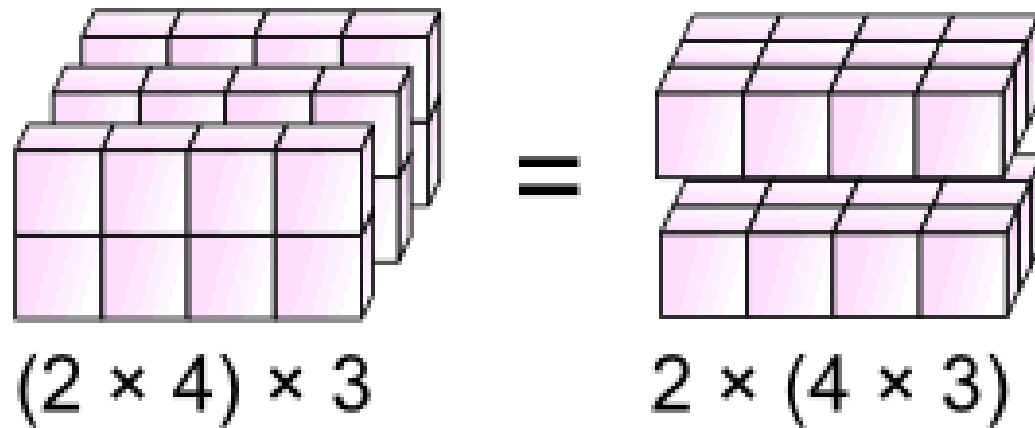


The diagram illustrates the Associative Law of Addition using colored dots. On the left, there are 6 blue dots, 3 orange dots, and 4 yellow dots, with the expression $(6 + 3) + 4$ below them. An equals sign follows. On the right, there are 6 blue dots, 3 orange dots, and 4 yellow dots, with the expression $6 + (3 + 4)$ below them.

$$(6 + 3) + 4 = 6 + (3 + 4)$$

Do you remember it?

Associative Law (2)



Do you remember it?

Kleene Star (1)

- Kleene star is a unary operation, either on sets of strings or on sets of symbols
- The application of the Kleene star to a set A is written as A^*
- Defined by Stephen Kleene in the context of regular expressions (will see this later in the course)

Kleene Star (2)

Given a set V we define:

$V_0 = \{\epsilon\}$ (the language consisting only of the empty string),

$V_1 = V$

$V_{i+1} = \{ wv : w \in V_i \text{ and } v \in V \}$ for each $i > 0$.

**Inductive
definition**

$$V^* = \bigcup_{i \in \mathbb{N}} V_i = \{\epsilon\} \cup V \cup V_2 \cup V_3 \cup V_4 \cup \dots$$

What do formal languages represent?

- A language is a **set of strings**
 - $L_1 = \{bc, ca\}$
 - $L_2 = \{ba, bb, bc, ca, cb, cc\}$
 - $L_3 = \{x \in \{a,b\}^* \mid (\exists y \in \{a,b\}^*) x = ay\}$
- How can sets of strings be applied in computer science?
 - Formal languages are not only mere mathematical representations

Languages in CS

- A language is a way of representing or communicating information
 - Not just meaningless strings
- There are many kinds of languages
 - Natural languages
 - Programming languages
 - Logic languages
 - ...

Example (1)

- Consider the following languages:
 - L_1 : set of “Word@Mac” documents
 - L_2 : set of “Word@PC” documents
- Operations:
 - L_1^c is set of documents that are not compatible with “Word@Mac”
 - $L_1 \cup L_2$ is the set of documents that are compatible with either Mac or PC
 - $L_1 \cap L_2$ is the set of documents that are compatible with both Mac and PC

Example (2)

- Consider the following languages:
 - L_1 : set of e-mail messages
 - L_2 : set of spam messages
- Operations:
 - $L_1 - L_2$ implements a filter

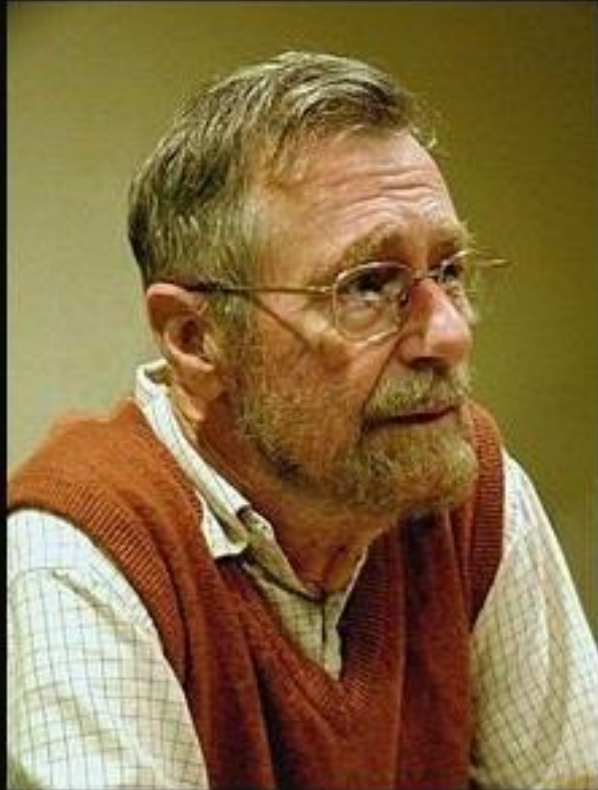
Languages in practice

- A language can represent
 - **Computations**
 - Documents
 - Programs
 - Multimedia
- **Operations on languages create new classes of languages**

Theoretical Computer Science

So far so good!

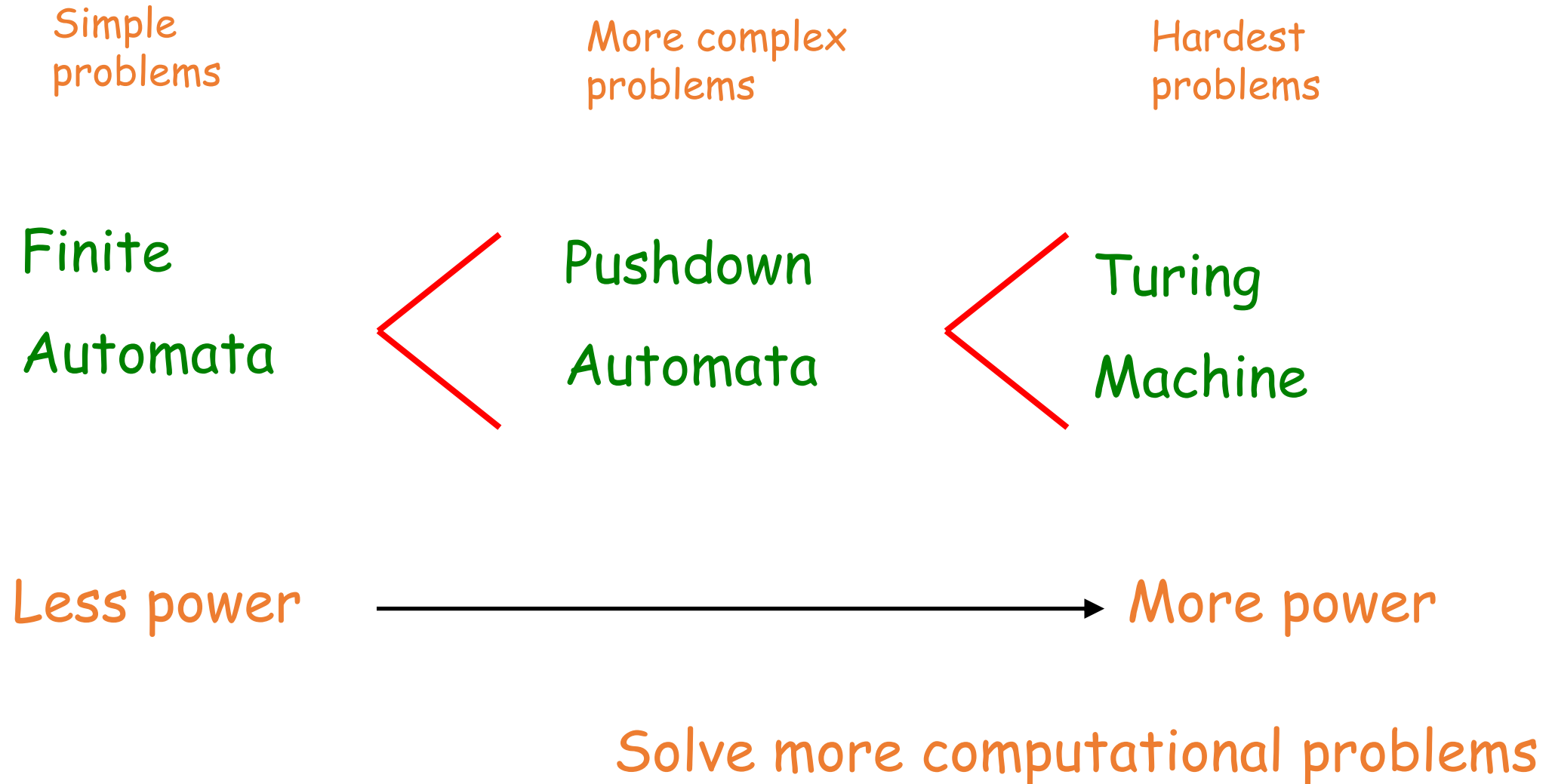
Lecture 3 - Manuel Mazzara



Computer science is no more about computers
than astronomy is about telescopes.

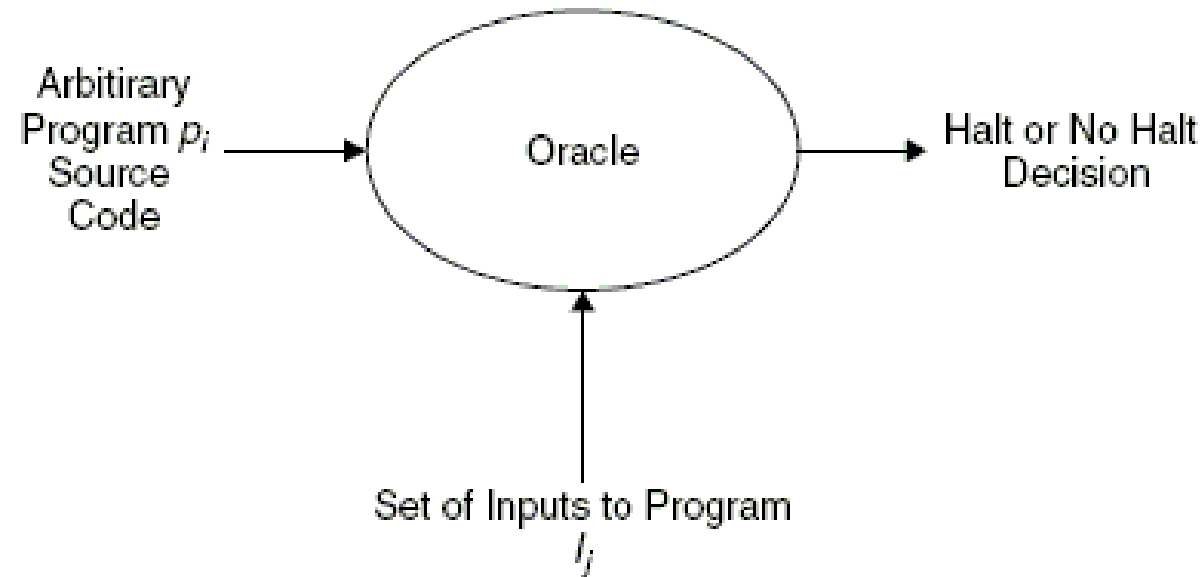
(Edsger Dijkstra)

Power of Automata



Our big question

- Is every function computable?
 - Can I write an algorithm for any function $N \rightarrow N$?
 - **Halting Problem**





From now on I will consider a language to be a set (finite or infinite) of sentences, **each finite in length** and constructed out of a **finite set of elements**. All natural languages in their spoken or written form are languages in this sense.

— Noam Chomsky —

An Alphabet!

And so are
Programming
Languages!

Theoretical Computer Science

Finite State Automata (FSA)

Lecture 3 - Manuel Mazzara

**A finite-state
automaton (FSA) is
a simple form of
imaginary machine**

AKA

- Finite-state machine
- Finite automaton
- Finite-state transducer
 - It is a variant that we will see
- Representations
 - transition table
 - **directed graph where vertices are states and edges are transitions**

FSA can have several purposes, we will mention

- ▼ some, but our main interest is **language recognition**

Why do I need FSAs?

- **FSAs are very useful and ubiquitous in computer science**
 - Compilers construction
 - Software design
 - Software verification
 - System specification...

Compilers Construction

- FSA is an imaginary machine which takes a **string of symbols as input** and changes its state accordingly
- FSA is a **recognizer for regular expressions** (and regular languages in general)
- **Lexical Analysis** within a compiler
 - A **lexer** goes character by character determining whether or not a word is an identifier, a keyword, or another token

Why do I need FSAs?

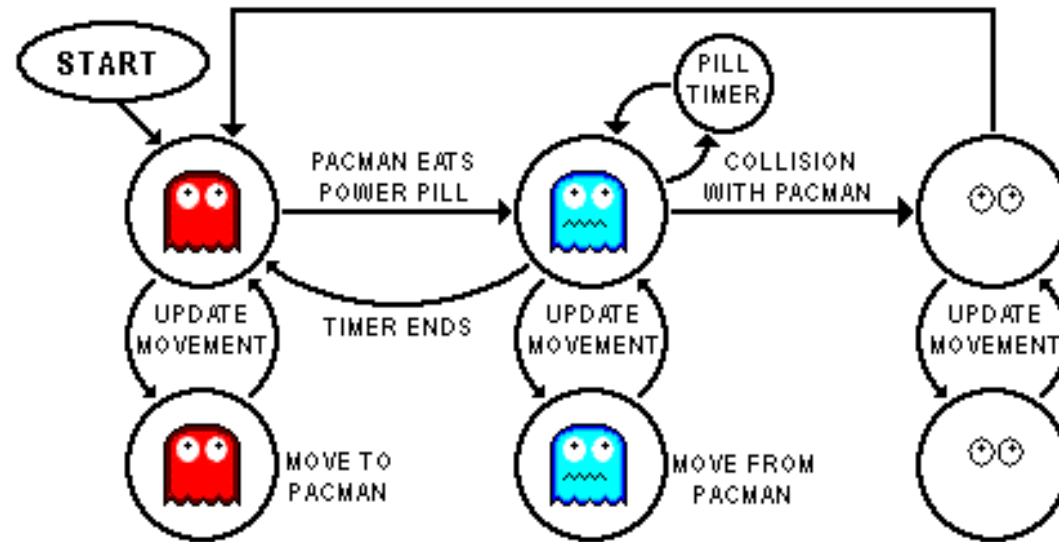
- Writing a simple compiler for a simple language **knowing no theory** is possible
- When languages becomes more complex there are **tools to abstract over** the process
 - A **lexer generator** produces a lexer for you without knowing too many details
 - For example **lex**, typical lexical analyzer generator on many Unix systems
 - You just need to **determine the regular expressions to identify**
- Emphasis on FSAs in compiler books is partly **historical**
 - **fast lexing used to be a huge problem** in the early computer science
 - the theory of FSAs is one of the earliest and most developed of computer science

Pac-Man

Released in Japan in 1980



Pac-Man Ghost



States

- An FSA has a **finite** set of **states**
 - A system has a **limited number of configurations**
- Examples
 - {On, Off},
 - {1,2,3,4, ...,k}
 - {TV channels}
 - ...
- States can be **graphically** represented as follows:



Input

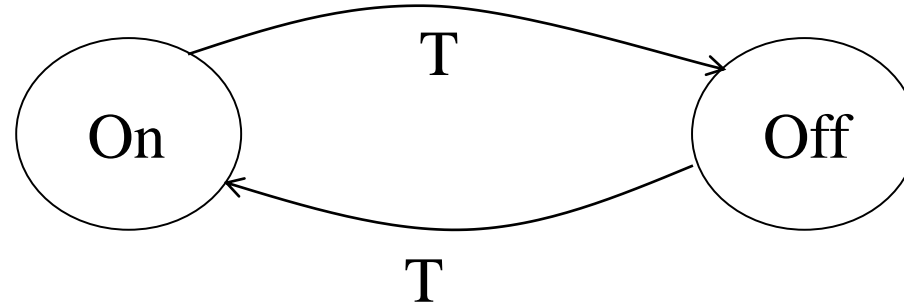
- An FSA is defined over an **alphabet**
- The symbols of the alphabet represent the **input** of the system
- Examples
 - {switch_on, switch_off}
 - {incoming==0, 0<incoming<=10, incoming>10}

Transitions among states

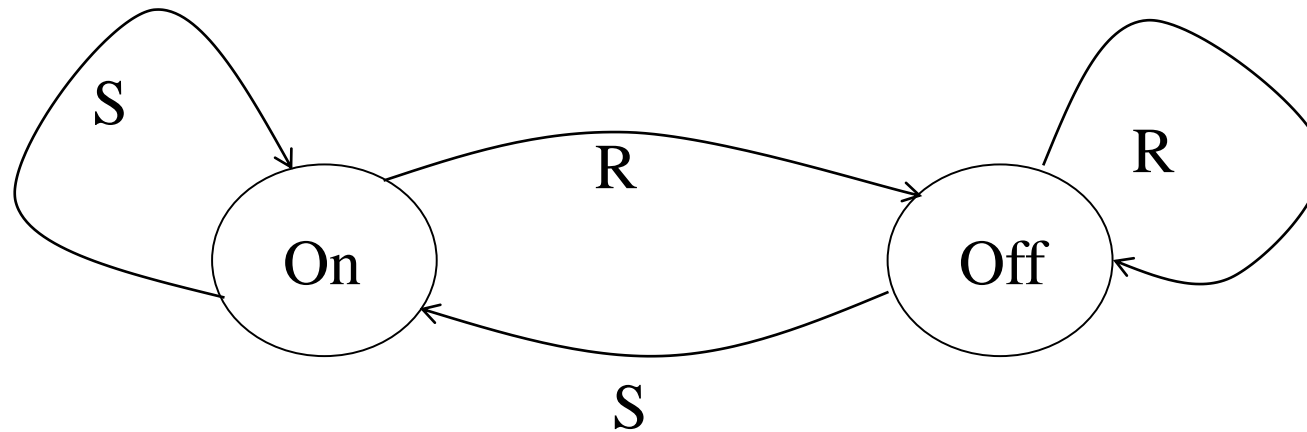
- When an input is received, the system **changes its state**
- The passage between states is performed through **transitions**
- A transition is graphically represented by arrows:



Simple examples



What are these two FSAs modelling?





FSA

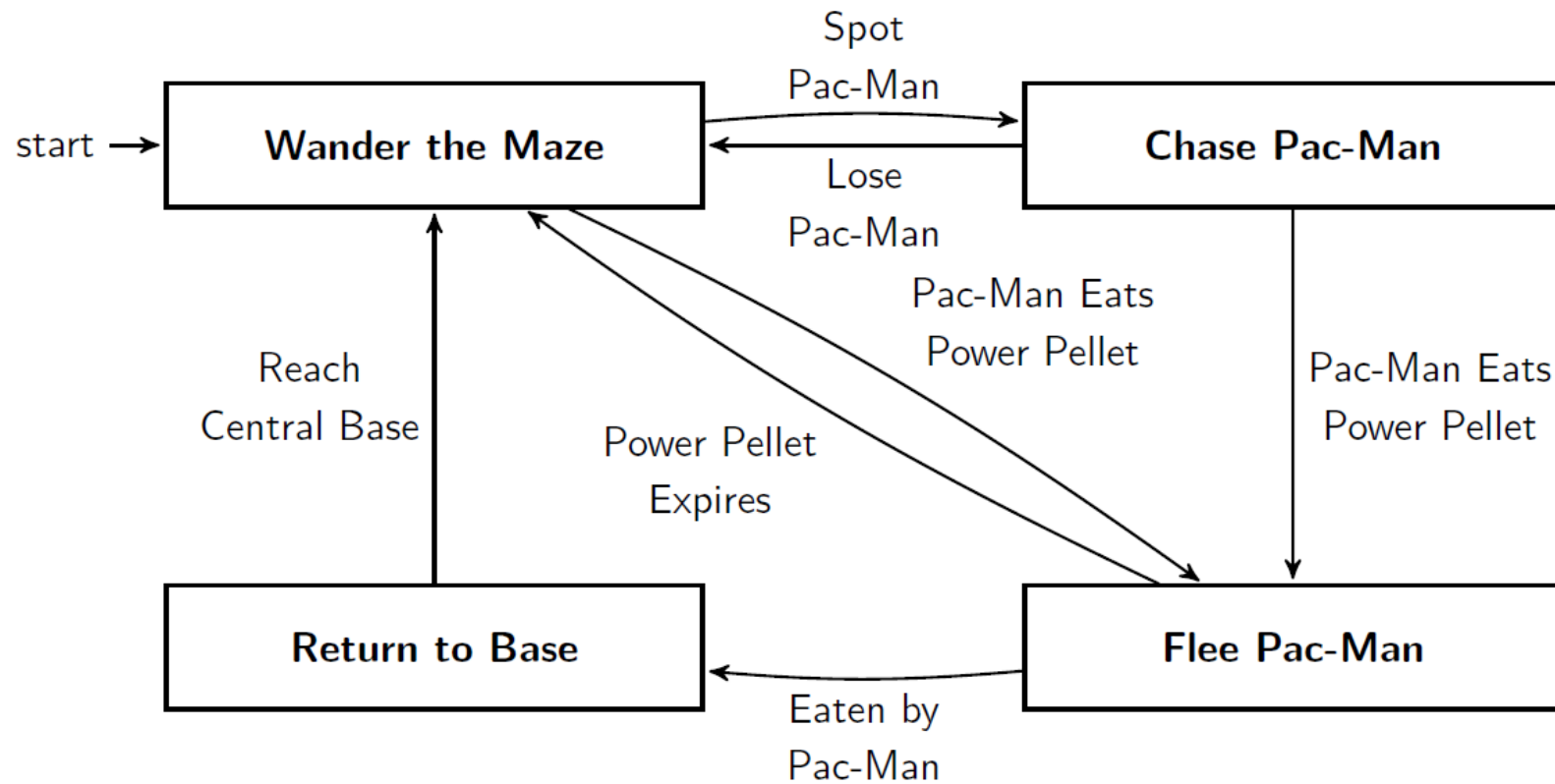
- FSAs are the **simplest** model of computation
- Many useful devices can be modeled using FSAs

... but they have some limitations

Applications of FSA

- Vending Machines
- Traffic Lights
- Video Games
- Text Parsing
- CPU Controllers
- Protocol Analysis
- Natural Language Processing
- Speech Recognition

Pac-Man Ghost again



History

- The world **automata** come from very far – we will see
 - Systems which **could transition between a finite number of internal states** have been known of for a long time
- Only relatively late in the early history of computer science they were formally studied
 - McCulloch&Pitts : A logical calculus of the ideas immanent in nervous activity (1943)
 - Kleene : equivalence of FSA and regexps (1956)
 - Mealy, Moore : generalized (1955-56)
 - Scott-Rabin : NDFSA (1959)
 - Ginsburg : Finite Transducers (1962)

FSA vs Turing Machine

Yes, I know we have not studied the TM yet!

- **Moore, Kleene** and others coming from different perspectives converged on the abstract model of the finite state machine characterized by its **input-output behaviour**
- A device of **finite nature**, differently from a machine with potentially infinite memory (Turing Machine), seemed **closer in structure to the digital computer**
- In 1950s the community of people interested in the theory of computation turned their attention to the device to understand possibilities and limits

- ▼ Turing showed what a computer **could NOT do**, even if given infinite time and space, but gave less insight on what a machine **could actually do**.

“Turing machines are widely considered to be the abstract prototype of digital computers; workers in the field, however, have felt more and more that the notion of a Turing machine is too general to serve as an accurate model of actual computers. It is well known that even for simple calculations it is impossible to give an a priori upper bound on the amount of tape a Turing machine will need for any given computation. It is precisely this feature that renders Turing's concept unrealistic.”

Rabin and Scott - "Finite automata and their decision problems," IBM Journal of Research and Development (April, 1959)

“In the last few years the idea of a finite automaton has appeared in the literature. These are machines having only a finite number of internal states that can be used for memory and computation. The restriction of finiteness appears to give a better approximation to the idea of a physical machine. Of course, such machines cannot do as much as Turing machines, but the advantage of being able to compute an arbitrary general recursive function is questionable, since very few of these functions come up in practical applications”


Rabin and Scott - "Finite automata and their decision problems," IBM Journal of Research and Development (April, 1959)

Theoretical Computer Science

Finite State Automata, formally

Lecture 3 - Manuel Mazzara

Math is easy,
notation may
overwhelm you!



As a computer scientist you must cope with notation – for example, programming

Informal vs. Formal

- Always three stages:
 - **Intuition**/idea/informal
 - **Examples**/instances
 - **Formal** definition
 - Human vs. machine understanding

Formally

- An FSA is (for now, to be extended) a triple $\langle Q, A, \delta \rangle$, where
 - Q is a finite set of states
 - A is the input alphabet
 - δ is a transition function (that can be **partial**), given by $\delta: Q \times A \rightarrow Q$

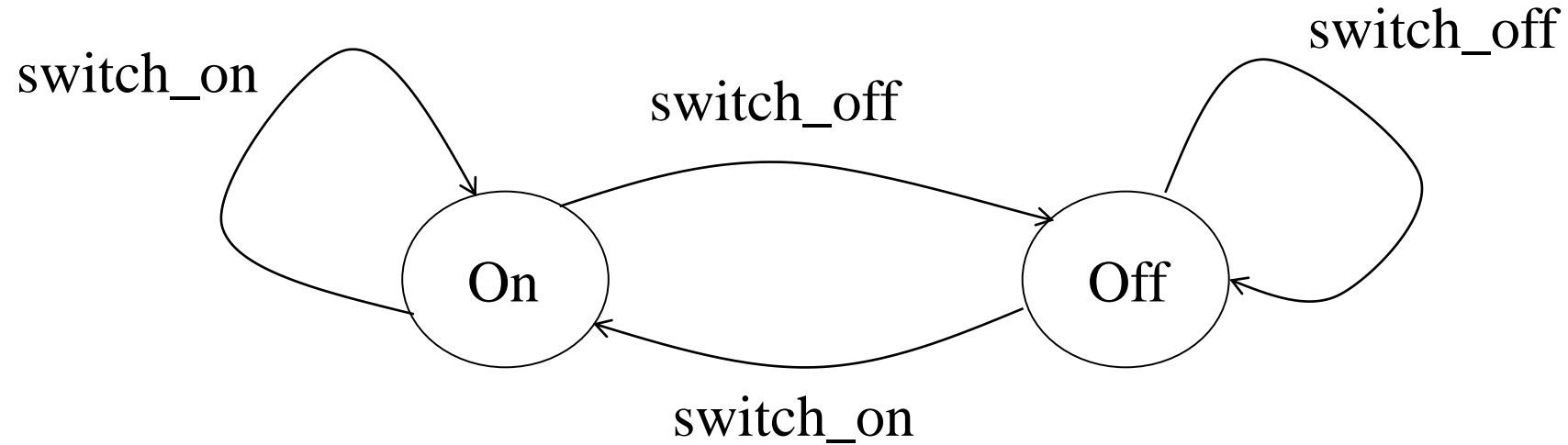


Delta (lowercase)

- Remark

if the function is partial, then not all the transitions from all the possible states for all the possible elements of the alphabet are defined (for example pressing sugar+ in a vending machine during coffee release)

Partial vs Total Transition Function



An FSA with a total transition function is called **complete**

Recognizing languages

- In order to be able to use FSAs for recognizing languages, it is important to identify:
 - the initial conditions of the system
 - the final admissible states
- Example:
 - The light should be off at the beginning and at the end

Elements

- The elements of the model are

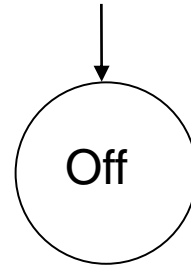
- **States**
- **Transitions**
- **Input**

and also

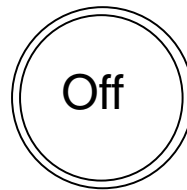
- **Initial state(s)**
- **Final state(s)**

Graphical representation

- Initial state



- Final state



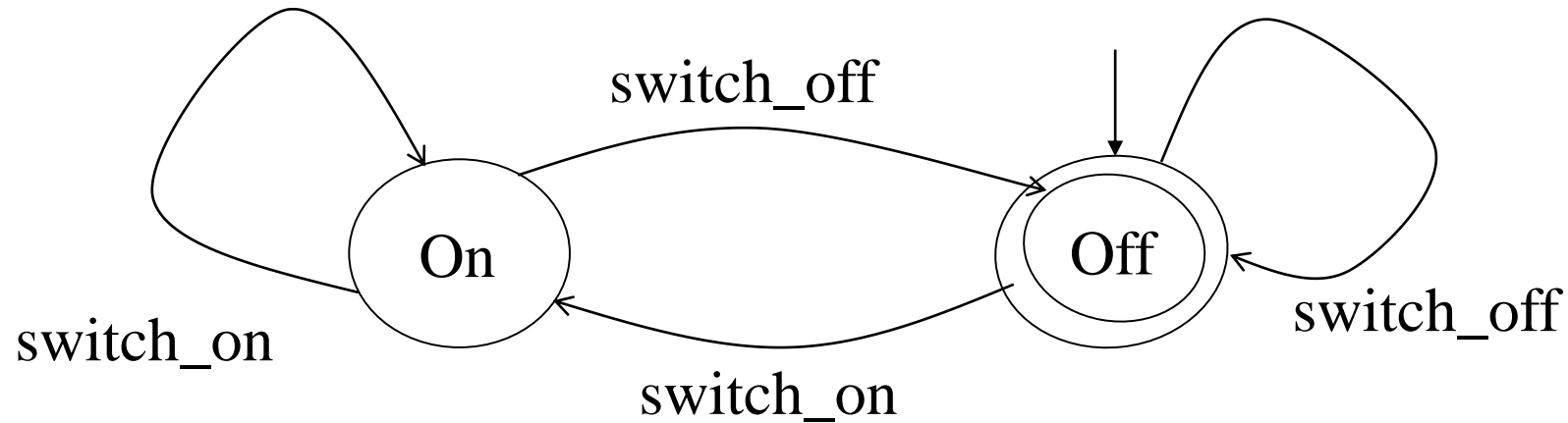
Formally

- An FSA is a tuple $\langle Q, A, \delta, q_0, F \rangle$ where
 - Q is a finite set of states
 - A is the input alphabet
 - δ is a (partial) transition function, given by
$$\delta: Q \times A \rightarrow Q$$
 - $q_0 \in Q$ is called initial state
 - $F \subseteq Q$ is the set of final states

In mathematics, a tuple is a finite ordered list (sequence) of elements

Move sequence

- A **move sequence** starts from an initial state and is ***accepting*** if it reaches one of the final states



Formally

- Move sequence:
 - $\delta^*: Q \times A^* \rightarrow Q$
- δ^* is inductively defined from δ
 - $\delta^*(q, \varepsilon) = q$
 - $\delta^*(q, y.i) = \delta(\delta^*(q, y), i)$
- Initial state: $q_0 \in Q$
- Final (or accepting) states: $F \subseteq Q$
- $\forall x (x \in L \leftrightarrow \delta^*(q_0, x) \in F)$

It is a recursive definition, we will see more about this next week!

A practical example

- Recognizing Pascal identifiers

