Automata and Computability

Discrete Event Systems

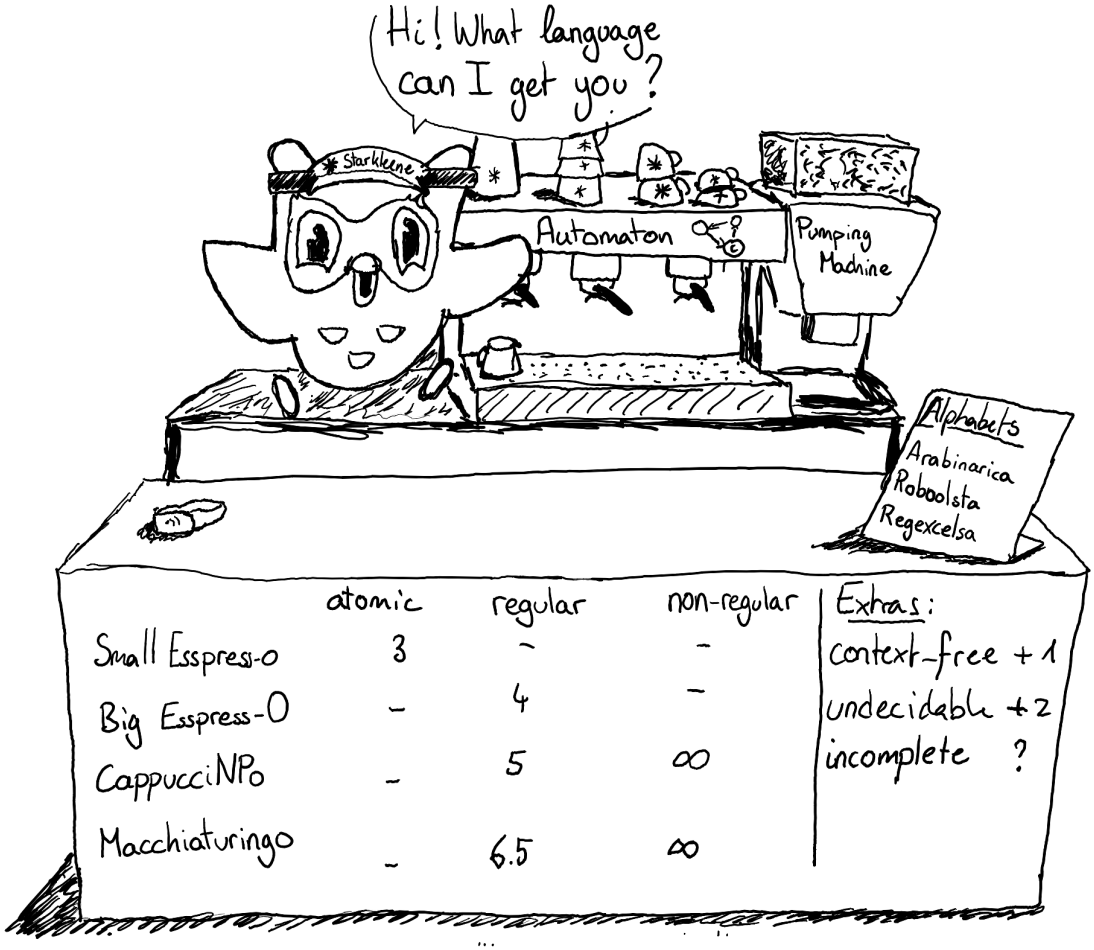

Figure 1: Duo never stops delivering new languages.

Disclaimer This manuscript is intended to support lectures about computability given in the course *Discrete Event Systems* lectured by Prof. Laurent Vanbever and the Networked Systems Group.
We emphasize that this is only supporting material, restating the important definition and ideas in a compact form as well as sketches for proofs (not complete proofs). Please refer to lecture material and references for more in depth explanations. Only the contents of the lecture slides and the exercises are required for the exam.
This is version 1.02, compiled on Wednesday 23rd September, 2026.
If you have ideas on how to improve this manuscript, please report any suggestions or issues to megretj@ethz.ch.

1 Automata Theory

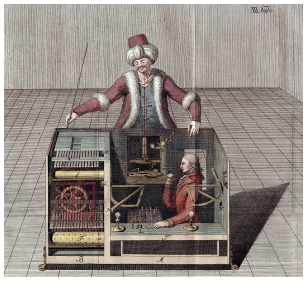

Figure 2: We will study very different and more abstract objects than the famous mechanical turk automaton. But in essence logical automata are also there to "mechanically" execute a purpose.

Automata theory allows the definition of models of computation, that is: what can a certain machine compute. The word automaton is inspired by physical machines.

> Wikipedia – An automaton (pl. automata, etym. "acting of one's own will") is a relatively self-operating machine or control mechanism designed to automatically follow a sequence of operations or respond to predetermined instructions.

Early computer scientists (back then still called mathematicians or logicians), including: Shannon, Ashby, Turing, Von Neumann, Moore, Kleene, Chomsky, ..., explored the idea of building logical tools to describe computation. Those models of computation are central to many modern applications such as text-processing, compiler design, programming-language design, artificial intelligence, and much more. We study two such models of computation: finite automata and context free grammars. This chapter is heavily based on Sipser's famous book[1].

[1] Michael Sipser. *Introduction to the Theory of Computation*. Thompson Course Technology, Boston, MA, third edition, 2013

2 Finite Automata

Def 1 (Alphabet, Strings and Languages)**.** An *alphabet* Σ is a set of symbols (characters, letters). A *string* or *word* over Σ is a sequence of symbols. The *empty string* ε is the string with no symbols. A *language* is a set of strings.

The term language here is to take in the broad sense of the term. It does not only refer to English, German, French,... or to Python, Rust, C, ..., but generally a way to convey meaning in a structured way. E.g. IP addresses are the language describing destinations on the internet.

We write $|S|$ to take the size of a set S. A set or language with a single element, $|S| = 1$ is *atomic*.

Some people talk of state-machines instead of automata, but really they're the same.

Def 2 (DFA)**.** A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set called the states
- Σ is a finite set called the alphabet
- $\delta : Q \times \Sigma \to Q$ is the edge transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accept or final states.

We write $M \in \text{DFA}$ to say that M is a DFA.

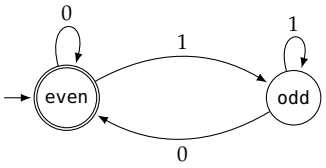

Figure 3: Example – a DFA that accepts even binary numbers when given from the most significant bit to the least significant one can be written as a tuple $(\{\texttt{odd}, \texttt{even}\}, \{0,1\}, \{(\_, 0) \mapsto \texttt{even}, (\_, 1) \mapsto \texttt{odd}\}, \texttt{even}, \{\texttt{even}\})$, or drawn as above.

We often represent automata as directed labeled graphs. Q are vertices (nodes), Σ are the possible edge labels, δ represents the outwards-going labeled edges (arrows) from a vertex, q_0 is where you enter the graph and F are the accepting vertices.

Def 3 (NFA)**.** A nondeterministic finite automaton (NFA) is like a DFA except:

- $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$

where $\mathcal{P}$ is the power set and $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

Def 4 (GNFA)**.** A generalised nondeterministic finite automaton (GNFA) is like an NFA except:

- There is only one accept state q_{accept}, i.e. $|F| = 1$.
- $\delta : (Q \setminus \{q_{accept}\}) \times (Q \setminus \{q_0\}) \to \mathcal{R}(\Sigma)$

where $\mathcal{R}(\Sigma)$ is the set of all regular expressions (def 13) over Σ.

By FA, we mean either a DFA, NFA or GNFA.

Figure 4: GNFAs are somewhere between REXs and NFAs

Def 5 (Recognized string)**.** A word w is *accepted* by a FA M (we write $w \vdash M$) iff $\exists$ a path starting at q_0 and labeled by w which ends in an accept state.

Def 6 (Recognized language)**.** The language recognized by an automaton M, denoted by $L(M)$, is the set of all strings accepted by M. That is, $L(M) = \{s | s \vdash M\}$.

2.1 Regular Languages

Def 7 (Regular language)**.** A language L is called regular, written $L \in$ REG, if a DFA recognizes it. That is, $\exists M \in \text{DFA} : L(M) = L$.

Thm 1. *All finite languages are regular. That is,* $|L| < \infty \Rightarrow L \in REG$.

Note that infinite languages may or may not be regular (see section 2.5).

Proof. If a DFA recognizes a language then it is regular, so here's how to construct a DFA with a finite set of words. Starting from an initial state, we create a chain for each word of the language ending in an accepting state. □

2.2 Operations on languages and Regular Operations

Operations on languages take some input language(s) and return an output language. We focus on five important operations: union, intersection, complement, concatenation and Kleene-star. To see what these operations do let's define what they do to DFAs. Let $M_1 = (Q_1, \Sigma, \delta_1, q_0^1, F_1), M_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$ be two DFAs operating on the same alphabet Σ.

Def 8 (Unioner – Cartesian Product Construction)**.** Define the unioner $M_\cup = M_1 \cup M_2$ as:

$$M_\cup = (Q_1 \times Q_2, \Sigma, \delta_\cup, (q_0^1, q_0^2), F_\cup)$$

where $F_\cup = Q_1 \times F_2 \cup F_1 \times Q_2$ and $\delta_\cup = \delta_1 \times \delta_2$, i.e. $\delta_\cup((q_a, q_b), j) = (\delta_1(q_a, j), \delta_2(q_b, j))$.

Def 9 (Intersector)**.** Define the intersector $M_\cap = M_1 \cap M_2$ as:

$$M_\cap = (Q_1 \times Q_2, \Sigma, \delta_\cap, (q_0^1, q_0^2), F_\cap)$$

where $F_\cap = F_1 \times F_2$ and $\delta_\cap = \delta_1 \times \delta_2$.

Def 10 (Complement)**.** Define the complement of M_1 as:

$$\overline{M_1} = (Q_1, \Sigma, \delta_1, q_0^1, Q_1 \setminus F_1).$$

Some operations are easier to describe using NFAs[2]. Let $N_1 = (Q_1, \Sigma, \delta_1, q_0^1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$ be two NFAs defined on the same alphabet and $Q_1 \cap Q_2 = \emptyset$, i.e. they are disjoint.

[2] but absolutely also possible to construct using DFAs!

Def 11 (Concatenation)**.** Define the concatenation $N_\bullet = N_1 \bullet N_2$ as:

$$N_\bullet = (Q_1 \cup Q_2, \Sigma, \delta_\bullet, q_0^1, F_2)$$

where $\delta_\bullet = \delta_1 \cup \delta_2 \cup \{(q_f, \varepsilon) \mapsto q_0^2 | q_f \in F_1\}$.

Def 12 (Kleene-star)**.** Define the Kleene-star of N_1 as:

$$N_1* = (Q_1^*, \Sigma, \delta_1^*, q_0^*, F_1^*)$$

where q_0^* is a new start state, $\delta_1^* = \delta_1 \cup \{(q_f, \varepsilon) \mapsto q_0^1 | q_f \in F_1^*\}$, $Q_1^* = Q_1 \cup \{q_0^*\}$ and $F_1^* = F_1 \cup \{q_0^*\}$.

Other operations are sometimes included but aren't necessarily as important and just syntactic sugar[3]. Amongst the above constructions, three of them are called *regular operations*. The reason we distinguish them from the rest, is that they are enough to generate all regular languages from the atomic languages of an alphabet with ϵ. Also, later we will see that they play different roles for context-free languages.

[3] For example, Kleene-plus $\{a\}+$ is the same as $\{\{a\} \bullet \{\{a\}*\}\}$.

Operation	Symbol	UNIX version	Meaning
Union	$\cup$	|	Match one of
Concatenation	$\bullet$	*implicit*	Match in sequence
Kleene-star	$*$	$*$	Match 0 or more times

Table 1: Regular expression operations on languages.

Thm 2 (Closure of regular operations)**.** *Regular languages are closed under regular operations.*

Proof. If $L \in$ REG then it can be written as a DFA M. Any DFA is by default also an NFA so we can use the above constructions to apply the operations and get NFAs. Finally, we can transform NFAs back to DFAs using the derandomization recipe in theorem 4. □

2.3 Regular Expressions

State-machines are nice to draw and reason about, but not very practical to type into a computer. Instead, to describe regular languages on machines we often use regular expressions. In section 2.4 we show that regular expressions and finite automata have the same expressivity.

Def 13 (REX)**.** Regular expressions (REX)[4] over an alphabet Σ are all expressions using regular operations over a set of atomic languages, containing: all singleton language of Σ, ε (the singleton language

[4] REX are often referred to as regex or regexp.
Notice that this definition is inductive: we define base elements and build from there on.

of the empty string) and $\emptyset$ (the language containing no words). The expressions are simplified using the operation preference order[5]: $* \prec \bullet \prec \cup$.

[5] $a \prec b$ means a precedes b

There are many different regex "dialects", including more or fewer operations. Commonly, complement $\bar{\cdot}$ and intersection $\cap$ are also used with preference order: $\bar{\cdot} \prec * \prec \bullet \prec \cap \prec \cup$.

Figure 5: Equivalence in one picture

[6] This is true because we defined these operations by operating on NFAs, see definitions 8, 11 and 12.

2.4 *Equivalence theorem*

The point is that regular languages can be expressed by any of DFA, NFA or REX and you can go from one to another.

Lem 3 (REX $\subseteq$ NFA)**.** $\forall r \in REX, \exists N \in NFA : L(r) = L(N)$

Proof. NFAs are closed under regular operations[6]. So, given a REX, transform each atom into an NFA and construct an NFA that simulates REX by applying the regular operations. □

Lem 4 (NFA $\subseteq$ DFA)**.** $\forall N \in NFA, \exists M \in DFA : L(N) = L(M)$

Proof. Derandomization recipe:

1. Set Q_M to the power set of Q_N.
2. Set F_M to be all states containing a state in F_N.
3. Make the nondeterministic transitions from N deterministic.
4. Translate the ε transitions into deterministic transitions.

□

NFAs have 3 types of non-determinism which the recipe solves: over-determinism, under-determinism and ε transitions.

Lem 5 (DFA $\subseteq$ REX)**.** $\forall M \in DFA, \exists r \in REX : L(M) = L(r)$

Proof. Use GNFAs to transform any DFA into a REX. □

Thm 6 (DFA $\sim$ NFA $\sim$ REX)**.** *DFAs, NFAs and REXs are equivalent.*

Proof. Follows immediately from lemmas 3, 4 and 5. □

Def 14 (Irreducibility)**.** A finite automaton is irreducible if all states are reachable from the start state (no useless states) and no two distinct states are equivalent (all outgoing transitions point to the same states).

2.5 *Pumping lemma*

The pumping lemma is a property about regular languages.

Thm 7 (Pumping lemma)**.** *If $L \in REG$, then $\exists p \in \mathbb{N}$ such that $\forall s \in L$ with $|s| \geq p$, s may be divided into 3 parts, $s = xyz$, and the following holds:*

1. $\forall i \geq 0, xy^iz \in L$
2. $|y| > 0$

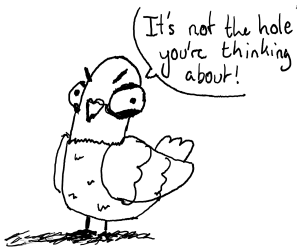

Figure 6: The pigeonhole principle: fancy name for a very boring observation.
[7] if $A \Rightarrow B$ then $\neg B \Rightarrow \neg A$

3. $|xy| \leq p$

This theorem bases itself on the pigeonhole principle; infinite languages can only be regular (i.e., described by a **finite** automaton, definition 7) if the automaton simulating them have a loop.

We use the pumping lemma as a way to prove that an infinite language is not regular by contrapositive[7]. The idea of disproving regularity is to show that for all $p \in \mathbb{N}$, there's at least one $s \in L$ with $|s| \geq p$ that cannot be divided in xyz such that the three properties hold. The pumping lemma says REG $\Rightarrow \exists p \in \mathbb{N}$, so by contrapositive $\nexists p \in \mathbb{N} \Rightarrow L \notin$ REG.

2.6 *Use-cases for finite automata*

Here are some well known applications for finite automata in computer science and how we use them in our research.

- Aho-Corasick algorithm string-searching algorithm. Used in Beeper an application-layer parser in eBPF to process IP-packets directly in the kernel.
- Levenshtein automata for string-searching with typos.

References

[1] Michael Sipser. *Introduction to the Theory of Computation*. Thompson Course Technology, Boston, MA, third edition, 2013.