TDDD14/TDDD85 Lecture 1: Introduction and Formal Languages

Victor Lagerkvist (based on slides by Christer Bäckström and Gustav Nordh)

In this lecture note we give an introduction to the fundamental concepts of the course, define the notion of a formal language, and the basic operations over these mathematical objects.

Introduction

Figure 1: A simple computer?

Figure 2: A more universal computer?

The heart of this course is to study models of computation. What can we compute with very limited memory, and what can we compute with an unbounded amount of memory? Consider e.g. the difference between a machine which recognizes a 4-digit numerical PIN code (Figure 1) and a full-blown personal computer (Figure 2). Is there a fundamental difference between these two devices?

While the simple machine in Figure 1 in principle could be implemented by a small, universal computer, it is not a great leap of faith to imagine that there might exist a simpler model of representation for the PIN code machine since it at any stage only needs to read a single digit as input and proceed accordingly, without needing to care about any past attempts. For example, suppose that the correct code is “1234”. Then the machine may immediately reject the current attempt if it reads a symbol outside the set $\{1, 2, 3, 4\}$, and reset the memory state. Similarly, if the machine is in its initial state, reads the symbol “1”, then it may immediately reject if the next symbol is not “2”.

In contrast, a universal computer certainly seems more complicated since it e.g. has access to additional memory which may be used to aid the computation. Another striking difference is that the PIN code machine — at least in principle — always terminates with a definite answer, but that it is certainly possible to write a computer program which does not terminate.

However, the title of this course is “formal languages and automata theory” and not “models of computation”, so what is the connection between these concepts? Given a computation device we typically want to use it to solve a given problem by computing something. As a theoretical convenience it is in this context furthermore common to only consider *decision problems*, which are idealizations of computational problems where it is sufficient to always answer either yes or no. For example, in the PIN code example we are given a string and want to answer YES if and only if the string is the correct PIN code 1234. Similarly, if we are writing a compiler for a computer language, then at some stage in this process we want to verify that a given program is syntactically correct, which we can also report by answering YES or NO.

More generally, if we have a machine M which always attempts to answer YES or NO to a given input, then the set of input strings for which the machine answers YES is known as the *language* of M and is in symbols typically written as $L(M)$. Thus, if M is the machine in Figure 1 which only accepts the PIN code 1234, then $L(M) = \{1234\}$. Hence, each computation device, an *automaton*, corresponds to a language, and our main objective is to describe the expressive power of different types of automata and study properties of the resulting languages. Let us now briefly describe the three main types of languages and automata that we will encounter in our course.

Finite Automata

A *finite automata* represents the simplest possible model of computation which is still powerful enough to result in interesting applications. This device consists of a finite number of "states" representing different stages of the computation, and for each symbol in the input string [1] proceeds to a new state through a set of transition rules. If there are no symbols left to read and the automaton reaches a so-called *accept state* then the machine answers YES, and otherwise NO. Crucially, a finite automaton has very little memory of the past, may not modify the input string in any form, and always terminates. See Figure 3 for a visualisation of a finite automaton. The PIN code machine from Figure 1 is an example of a finite automaton.

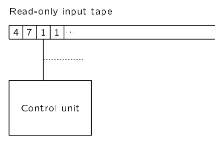

Figure 3: A visualisation of a finite automaton.

[1] Sometimes visualized as an old-school magnetic tape.

The corresponding languages are so-called *regular languages*, and exactly corresponds to languages describable by *regular expressions*, which has an abundance of applications in computer science.

Pushdown Automata

The pushdown automaton is a generalisation of finite automata where the machine is equipped with additional memory in the form of a stack, where symbols may be pushed and pulled. Thus, a pushdown automaton can store an unbounded amount of items, but only within the confinements of a stack (e.g., we only have access to the topmost element). For example, assume that we in the context of a parser want to be able to recognise whether a given string of parentheses is *balanced*, i.e., each left parenthesis (has a matching right parenthesis) later in the string. This language is known to *not* be regular but can easily be recognised by a pushdown automaton by

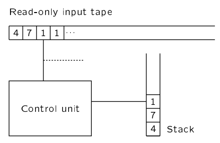

Figure 4: A visualisation of a pushdown automata.

pushing and pulling in an appropriate way depending on whether the current input symbol is (or). See Figure 4 for a visualisation of a pushdown automaton.

The corresponding class of languages is the set of *context-free languages* and is a much richer class of languages than regular languages. An important application may be found in parsing of programming languages where it is common to describe the language in question by a context-free grammar, and then parse the language by using algorithms inspired by pushdown automata.

Turing Machines

The most powerful model of computation that we consider is the *Turing machine*, named after the British mathematician Alan Turing. The difference between a Turing machine and the two previously mentioned classes of automata is that a Turing machine may not only inspect the input string, but also (1) modify it and (2) make it longer so that the Turing machine in effect also has unbounded memory (see Figure 5). This seemingly minor modification results in a significantly increased expressive strength, and a Turing machine is believed to be able to compute *everything* that can be computed, a conjecture known as the *Church-Turing thesis*. Modern computers, including smartphones and similar devices, all operate in a fashion similar to Turing machines. The main advantage of studying Turing machines instead of concrete computers is that they are much simpler to describe and reason with, which in turn makes it easier to prove mathematical properties.

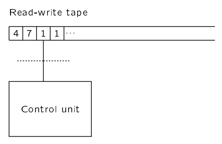

Figure 5: A visualisation of a Turing machine.

Formal Languages

We now properly begin the first topic of the course. Thus far we have seen examples of computation devices, automata, and argued that each computation device corresponds to a language. But what, precisely, is a formal language? Let us try to answer this question before we continue by studying classes of automata. Our first definition is that of an *alphabet*, which behaves similarly to an alphabet in many natural languages, in the sense that it is used as building blocks to form words.

Definition 1. *An* alphabet *is a* finite *set of symbols (typically denoted by* Σ*).*

The condition that an alphabet is finite is crucial[2]. Otherwise, given a symbol, how could we verify that the symbol is included in

[2] Note that Σ is allowed to be the empty set $\emptyset$. We could explicitly have forbidden the empty set to be an alphabet but this would lead to a less elegant definition.

an alphabet? The definition of a *string* over an alphabet then follows naturally.

Definition 2. *A* string *over an alphabet* Σ *is a* finite *sequence of symbols over* Σ*, i.e., each symbol is included in* Σ.

A *formal language*, or simply *language*, is then just a set (finite or infinite) of strings over a given alphabet Σ. However, before we turn to properties and operations over languages we will define some important operations over strings.

Example 1. *Consider the following strings and alphabets.*

- 1010 *is a string over* $\Sigma_1 = \{0, 1\}$.
- 1234 *is a string over* $\Sigma_2 = \{1, \ldots, 9\}$.
- theory *is a string over* $\Sigma_3 = \{a, b, \ldots, z\}$.
- $\mathrm{fact{:}theory}$ *is a string over* $\Sigma_4 = \{\mathrm{fact}, \mathrm{theory}, \mathrm{lemma}, \mathrm{proof}, :\}$.

Note that we in the last three examples did not require each symbol in the alphabet. This is perfectly fine since the condition in Definition 2 only requires that each symbol is included in the alphabet.

Definition 3. *The* length *of a string* x*, written* $|x|$*, is the number of symbols in* x.

The only aspect of this definition which might be puzzling is that "the number of symbols in x" depends on the alphabet in question.

Example 2. $|1010| = |1234| = 4$ *(with respect to* Σ_1 *and* Σ_2*),* $|\mathrm{theory}| = 6$ *(with respect to* Σ_3*), and* $|\mathrm{fact} : \mathrm{theory}| = 3$ *(with respect to* Σ_4*).*

Definition 4. *The* empty string *(*ε*) is the string of length* 0.

The empty string plays roughly the same role as the number 0 in arithmetics and the empty set $\emptyset$ in set theory. Let us proceed by defining a handful of additional operations over strings.

Definition 5. *The* concatenation *of* x *and* y *is written* xy.

Note that the empty string concatenated to any string x does not result in a new string, i.e., $\varepsilon x = x\varepsilon = x$.

Example 3. *If* $x = \mathrm{red}$ *and* $y = \mathrm{fox}$ *then* $xy = \mathrm{redfox}$.

Naturally, we may also concatenate a string x with itself, and this operation is important enough to warrant a definition.

Definition 6. *Let* x *be a string.*

- $x^0 = \varepsilon$*, and*

- $x^k = \overbrace{x \ldots x}^{k\text{times}}$ *for* $k \geq 1$.

Example 4. *If* $x = 01$ *and* $y = 00$ *then* $x^3 y = 01010100$.

Definition 7. *Let* Σ *be an alphabet. We let* $\Sigma^* = \{x \mid x \text{ is string over } \Sigma\}$ *be the set of strings over* Σ.

As a convention we let $\varnothing^* = \{\varepsilon\}$ (recall that an alphabet is allowed to be empty)[3]. We now have sufficient machinery to continue the study of formal languages. It may be illuminating to see that the notion of a formal language may concisely be defined via Σ^*.

[3] Trivia: if you have previously taken a course in abstract algebra, then it might be interesting to note that Σ^* with concatenation and unit element ε forms a *monoid*, since the concatenation operator is associative. This explains why the empty string ε behaves similarly to 0.

Definition 8. *Let* Σ *be an alphabet. A set* $X \subseteq \Sigma^*$ *is called a* formal language, *or simply a* language.

Example 5. *If* $\Sigma = \{0, 1, \ldots, 9\}$ *then the following sets are all examples of formal languages over* Σ.

1. $\varnothing$,
2. $\{\varepsilon\}$
3. $\{9, 10, 11\}$,
4. $\{1234\}$,
5. $\Sigma^* = \mathbb{N}$ *(the set of all natural numbers).*

Note that $\varnothing$ *(the set without elements) and* $\{\varepsilon\}$ *(the set containing the empty string) are two different sets, but both are languages over* Σ *since (1)* $\varnothing$ *is a subset of every set and (2)* Σ^* *by definition contains* ε *since* ε *trivially is a string over* Σ.

Given two languages A and B over an alphabet Σ there are many natural ways to combine A and B to obtain a new language over Σ. For example, we could produce the set containing all elements which are elements of both A and B (*intersection*), or the set consisting of elements included in either A or B (*union*). Formally, we define these operators, and a handful more, as follows.

Definition 9. *Let* A *and* B *be languages over an alphabet* Σ. *We define the following set operations.*

1. $A \cap B = \{x \mid x \in A, x \in B\}$ *(intersection).*
2. $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$ *(union).*
3. $\bar{A} = \{x \in \Sigma^* \mid x \notin A\}$ *(complement).*
4. $\mathcal{P}(A) = 2^A = \{X \mid X \subseteq A\}$ *(powerset).*

5. $AB = \{xy \mid x \in A, y \in B\}$ *(concatenation).*

6. • $A^0 = \{\varepsilon\}$, *and*
   • $A^k = \overbrace{A \dots A}^{k \text{ times}}$ *for* $k \geq 1$ *(the kth power).*

7. $A^* = A^0 \cup A^1 \cup A^2 \cup \dots$ *(the star operation).*

8. $A^+ = AA^* = \bigcup_{n \geq 1} A^n$ *(union of all non-zero powers of* A*).*

The star operation A^* is by design very similar to the operation Σ^* where Σ is an alphabet, and the only difference between these two concepts is that A in Definition 9 is allowed to be infinite.

Example 6. *Consider the following examples over* $\Sigma = \{0, 1\}$.

1. $\{0,1\}^0 = \{\varepsilon\}$.
2. $\{0,1\}^1 = \{0,1\}$.
3. $\{0,1\}^2 = \{00, 01, 10, 11\}$.
4. $\{0,1\} \cup \{\varepsilon\} = \{0, 1, \varepsilon\}$.
5. $\{0,1\} \cap \{\varepsilon\} = \emptyset$.
6. $\{0,1\}^2\{\varepsilon\} = \{0,1\}^2$.
7. $\{0,1\}^k = \{x \in \{0,1\}^* \mid |x| = k\}$ *(i.e., the set of Boolean strings of length k).*
8. $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$ *(all strings over* 0 *and* 1*),*
9. $\{0,1\}^+ = \{0, 1, 00, 01, 10, 11, 000, 001, \dots\}$ *(all non-empty strings over* 0 *and* 1*),*
10. $\{0,1\}^* \cap \{0,1\}^k = \{0,1\}^k$.
11. $\{0,1\}^{**} = \{0,1\}^*$.
12. $\{0,1\}^* \cap \emptyset = \emptyset$.
13. $\{0,1\}^* \cap \{\varepsilon\} = \{\varepsilon\}$.

Summary

After giving a brief introduction to the course we defined (1) strings and operations on strings, and (2) languages and operations on languages. This is a reasonable starting point but we have opened up more questions than we have answered. In the next lecture we will formally define our first computation device, finite automata, and investigate the resulting class of languages.

Food for Thought

1. Is the set of *rational* numbers $\mathbb{Q}$ (e.g., $1, \frac{1}{2}, \frac{4}{7}, \ldots$) a formal language?
2. Is the set of *real* numbers $\mathbb{R}$ (e.g., $1, \frac{1}{2}, \pi, 2, \ldots$) a formal language?
3. Is written English, as it is normally understood, a formal language?