Mathematical Foundations of Formal Language Theory

Alexander Meduna, Lukáš Vrábel, and Petr Zemek

Faculty of Information Technology, Brno University of Technology

January 6, 2013

Alexander Meduna, Lukáš Vrábel, and Petr Zemek

Mathematical Foundations of Formal Language Theory

January 6, 2013

Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic

Abstract

This document gives a gentle introduction into the mathematical foundations of formal language theory. More specifically, it reviews sets, relations, sequences, functions, relational closures, types of mathematical statements and their proofs. To explain these basic mathematical notions that underly the theory of formal languages, we use a running example, where, starting solely from the idea of a finite automaton, we gradually build our way up to a formal definition of this model. In this way, it can be seen not just how various mathematical concepts are applied, but also the motivation for using them over the others. We also cover the basics of mathematical statements and their proofs. The present document emphasises clarity and comprehensibility. Every notion is described informally prior its definition, and many examples illustrating the concepts are provided.

Acknowledgements

The present work was supported by the FRVŠ MŠMT FR271/2012/G1 grant. We wish to thank our girlfriends, Ivana, Pavla, and Daniela, whose everlasting interest and constant support have given us the motivation to realize this work.

Contents

1 Introduction

Have you ever wondered why there is so much mathematics in computer science? For example, computer graphics is filled with linear algebra, relational databases are built upon relations, computer networks greatly utilize graph theory, compiler design take advantage of automata, and the area of formal verification and analysis use formal methods. Indeed, everywhere you look, you see the "queen of science". Some people even say that computer science in general is mathematics' younger sister. Why is this so? In this document, we will show why mathematics is important for computer science, and how to understand, read, and write it.

However, the area of computer science is so huge that we cannot cover it all. Instead, we narrow our main attention to formal language theory, which fulfils a crucial role in most computer science fields that grammatically analyze and process languages, ranging from compilers through computational linguistics up to bioinformatics. We will see which mathematical concepts fit very nicely in this theory and the reasons for using them.

One of the basic mathematical concept is *formalization*. We take a real-world scenario and convert it into a formal mathematical representation. Then, we may formally deduce properties of this representation and actually prove that the system works as intended by using one of the strongest weapon for reasoning—mathematical logic. Thus, we can connect and use one of the oldest sciences known to man and profit from its vast knowledge base.

Consider the following three real-life scenarios. Mainly, notice their similarities, which we will discuss shortly.

1. *Coke-vending machine.* Consider a classical coke-vending machine which, given a sufficient amount of money, provides a beverage. Such a machine may provide a variety of different beverages as well as it may accept coins or paper money. After choosing what you want and putting enough money into the machine, you will receive the chosen drink. Furthermore, if you overpaid, you will (in most cases) get the change back. These machines start with no beverage selected and no inserted money. Then, as the customer provides input in the form of drink selection and money, the machine changes its state reflecting these inputs. Finally, after a drink is provided, it restarts into the start state.
2. *Soldier in a computer game.* Apart from their look, soldiers from computer games, like Warcraft or Operation Flashpoint, differ in many ways. For example, they may

hold various equipment, like weapons, have different skills, and fulfil different tasks. Their current behavior is altered by changes in the environment. For example, if an enemy is in sight, the soldier may switch to an aggressive state and start shooting at him, or, when the enemy is too powerful, he can switch to a fleeing state and run away from him. Usually, a soldier starts from some default state. Then, as the game evolves, he is switching to various other states based on the environment and commands, and finally ends if he dies or the end of game is reached.

3. *Removal of comments from computer programs.* Consider, for example, the block comments in the C language, delimited by "`/*`" and "`*/`". Usually, a tool that removes such comments from C source code is a part of compilers. It reads the input program in a symbol-by-symbol way, and when encountering "`/*`", it changes its state to a comment-removing state so that the comment does not appear in the output. Furthermore, it has to remember when it is inside of a string, where comments are not to be eliminated. Indeed, "`printf("/* Hello */ Ben")`" should print "`/* Hello */ Ben!`", not just "` Ben`". A tool like this usually starts at the beginning of the input program and ends when the end of the input file has been reached.

Naturally, engineers get into contact with these kinds of situations. When designing a computer system or program used to simulate or solve problems in such scenarios, it is useful to find some simple formal system preserving all the important properties of the scenario. Having such systems allows us to answer a lot of questions more easily. The first step in finding such a system is to find these important properties.

From a rather general viewpoint, all of the above scenarios share the following five similarities:

- First of all, they have some *state* by which they can be specified in any time moment. The state of a coke-vending machine is composed of the selected beverage and the amount of money that have already been put into the machine. A soldier may be described by his equipment, set of skills, goal, position, etc. Finally, the tool for eliminating block comments in C code may be specified by the piece of information inside which syntactical construct it appears (block comment, string, etc.) and by the unread part of the input.
- They act based on the *input* provided from the environment. In the case of a vending machine, the input are choices of the user (type of beverage) and inserted money. The act of a soldier depends on the presence of enemies, items, and other things that surround him. And, as for the comment-eliminating tool, the input are, of course, the read characters that the source code is composed of.
- The change of their state is based on the current state and the currently processed input. This change may be prescribed by so-called *state transition rules.* When we insert a new coin into a vending machine, it has to remember this fact by switching its state. Soldiers work in the same way. For example, if a soldier has a gun (state) and sees an enemy (input), it starts to fire at him (the next state). In the source-code-altering tool, we may prescribe that after reading "`/*`", it starts to discard the subsequent characters until "`*/`" is read.
- They all start from some designated *start state.* A coke-vending machine usually starts with no pre-selected beverage and no inserted coins. A soldier may start from some default state, like without any equipment and waiting for orders. And, as we have already said, our tool starts at the beginning of the input source code.

- They end when they reach some *final state*. In the case of a vending machine, such states may be all states where we have selected a beverage and have inserted a sufficient amount of money. The number of such states depend on the variety and prices of products as well as on the permissible types of money. A soldier may end when he dies, finishes his assignment, or the end of the game has been reached. The comment-eliminating tool ends when the input source program has been completely read and transformed.

These similarities bring us to the idea of creating a single simple model for all of these scenarios. This concept is described next.

The main strategy behind the usage of mathematics for problem solving is a process called *abstraction.* The word *abstraction* is used in the following sense: we take these somewhat complex real-world problems, and we simplify them as much as we can—that is, we strip them of everything that can be left out, everything that is not important for our problem solving. For example, in the case of a coke-vending machine, it is not really important if the drink is a coke. It is not even important that it is a drink, or that we are counting coins! The only thing we need to concern ourselves with is that we need to add up some numbers and then do some action.

After we simplify as much as we can, we are left with a very simplified model of real-world problems. This model has, in fact, almost no resemblance to the original real-world case study. We do this simplifying in order to solve the key problems while doing as little work as possible as well as minimizing the chance of errors. We can then focus just on the important parts of our problem and we do not have to concern ourselves with tedious petty details.

Actually, there is an even better approach—we can even entirely skip the model creation part! We can just simplify until we reach an existing abstract model, and then we can use previously established results of other people as the basis for our solution to the problem. In our case, the very well understood model of a *finite automaton* is a simple model that is sufficient for our problems. Of course, for some problems we could simplify even more, and some of the problems will need a little bit extra complexity for the best solutions. Thus, in order to achieve the *best* solutions, we could create an entirely new model for every problem. However, by using the theory of finite automata, we can build from the already known results[1]. By using these known results, we will save a lot of time and money, which would be otherwise spent to reinvent the “wheel” in slightly different, new model, and we will get much better *efficiency.* Indeed, if we look at the model of a finite automaton intuitively, it consists of a lot of different states, and the active state is changed based on the inputs of the automaton. That is everything we need for our scenarios.

Furthermore, by learning mathematics, abstraction, and model creation, you are learning one of the core skills of an engineer. Sometimes, as can be seen in Figure 1.1, computer-science students are more inclined to learn about popular, hot topics. However, these attractive topics tend to be flushed away, being replaced by other seasonal booms. On the other hand, core topics in computer science provide firm foundation on which one can build and learn.

[1] One of such useful results would be that for implementing a finite automaton of n states, we need just a simple electronic circuit with $\log_2 n$ memory bits.

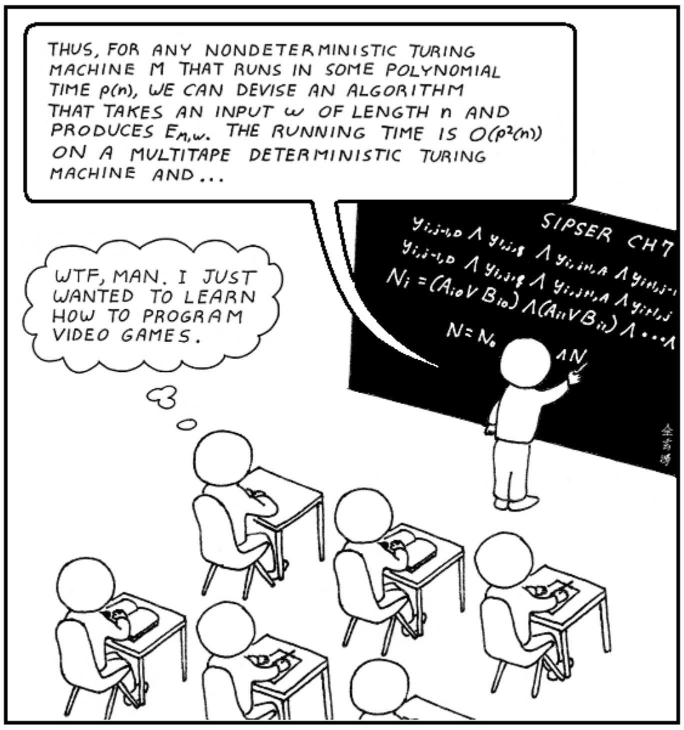

THUS, FOR ANY NONDETERMINISTIC TURING MACHINE M THAT RUNS IN SOME POLYNOMIAL TIME p(n), WE CAN DEVISE AN ALGORITHM THAT TAKES AN INPUT w OF LENGTH n AND PRODUCES E_{M,w}. THE RUNNING TIME IS O(p²(n)) ON A MULTITAPE DETERMINISTIC TURING MACHINE AND...
WTF, MAN. I JUST WANTED TO LEARN HOW TO PROGRAM VIDEO GAMES.
SIPSER CH7

Fig. 1.1. Sometimes, students are more inclined to learn about popular topics in contrast to core ideas of computer science (adopted from [10]).

Example 1.1. Examples of some hot topics for computer science in early 2000, compared to the corresponding underlying core topics:

Hot topic: cloud computing
Core topics: distributed systems, distributed algorithms, operating systems

Hot topic: programming in C♯
Core topics: programming language theory, compilers

Hot topic: multi-core systems
Core topics: computer architecture, instruction sets, digital systems

Hot topic: writing video games
Core topics: graphics, linear algebra, digital image processing, artificial intelligence

Learning math and core topics allows students to gain new skills in the future more easily. On the other hand, learning hot topics without understanding the underlying core ideas will probably make students unprepared for the things of the near future.

Goals

The principle goal of this document is to give you a gentle introduction into the mathematical foundations of formal language theory. The secondary goals include a demonstration of the usefulness of mathematical notation and rigorous methods, notes on how to read and write your of definitions and statements, and why are mathematical proofs of major importance.

Approach

To explain the basic mathematical notions that underly the theory of formal languages, we will use a running example. Starting solely from the idea of a finite automaton, we will gradually build our way up to a formal definition of this model. In this way, you will not just see how various mathematical concepts are applied, but also the motivation for using them over the others. Of course, during the course, we will also provide many worked-out examples to illustrate the concepts. Furthermore, we will try to relate the concepts to an area of application of formal language theory—programming.

Prerequisites

Although this text is self-contained in the sense that no other sources are needed to grasp all the presented material, the reader is expected to have at least basic knowledge regarding elemental notions from formal language theory, like strings, languages, and operations over them.

Organization

The present text is organized as follows. After this introductory chapter, Chapter 2 guides you through a mathematical formalization of finite automata. In a greater detail, it provides you with the basics concerning sets, sequences, relations, and functions. At the end of the chapter, we put all of these notions together to create a formal definition of a finite automaton. Next, in Chapter 3, we introduce more notions that are needed to properly and formally define a computation and the accepted language of a finite automaton. After that, Chapter 4 covers the basics of mathematical statements and proofs. As mathematics is based upon statements like theorems and their proofs, it is vital for you to understand the motivation behind proving and stating proved results formally. Chapter 5 concludes the present text by giving a summary of everything you have learned during our journey through the world of mathematical foundations of formal language theory.

2

Defining Finite Automata

In the introduction, we have seen how a simple formal model—*finite automaton*—is suitable for modelling many real-world scenarios. Basically, a finite automaton is composed of *states* and *rules* that are used to transition between these states. In essence, states represent information and rules are used to modify this information based on the input from the environment. Usually, a single state is designed to be the *start state.* From this state, the automaton always starts its computation process. Furthermore, some states are marked as *final states.* They tell us that in such states, the automaton can successfully finish its computation. Every transition rule has associated a symbol, which represents a piece of information that is obtained from the environment. The automaton then works in the following way. It starts in the start state. Based on the information provided from the environment, it selects an appropriate rule which will take him into another state. For every such piece of information, it uses a rule which results into the change of the current state. When the automaton appears in a final state and there is no more information from the environment, the computation is *successful.* Otherwise, if there is either no appropriate rule to be used or no more information and the automaton has not reached a final state yet, the computation is said to be *unsuccessful.*

A very succinct and illustrative way of representing finite automata is their *graphical representation.* It is of the form of a *transition diagram.* This diagram shows the states of the automaton, the rules used to transition from a state to another state, the designated start state and final states. In Figure 2.1, we see an example of a finite automaton representing a rather simplified version of a payment mechanism in a coke-vending machine.

In such a diagram, states are represented by round rectangles whose label is a symbolic name of the state. In our example, we label states by the amount of money that was already put into the machine and the amount of money the machine will return as change. For simplicity, let us imagine that we have only coins with values `2` and `5`, and a coke costs `4`. Then, the state labeled by $\langle 0 \rangle$ denotes the fact that when the automaton is in this state, no money has been put into the vending machine. States labeled by $\langle 2 \rangle, \langle 4 \rangle, \langle 5 \rangle$, and $\langle 7 \rangle$ represent the given sum of coins that have been put into the vending machine. Furthermore, states whose labels start with `ret:` represent the final states of the automaton. In these states, enough money have been already put into the vending machine in order to get a coke. In addition, these states also store the information about the amount of money that have to be returned. The start state is denoted by an ingoing

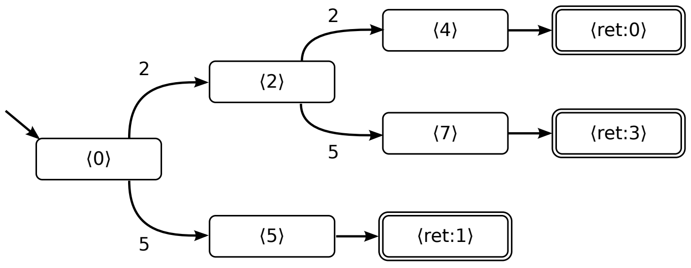


Fig. 2.1. A transition diagram of a finite automaton representing a payment mechanism in a coke-vending machine. When the price of the coke is 4, some states are needed to sum up the coins while other states store the amount of change returned to the customer along with the coke.

arrow, and final states are marked by double outline. Regarding transition rules, they are written as arrows from a state into another state, labeled by a piece of information provided by the environment. In our case, the environment is the person who wants a coke, and the pieces of information are the coins which are put into the machine. You can see that some arrows have a coin value attached to them, with corresponding sums in the adjacent states. On the other hand, some arrows have no additional symbols. These arrows represent transitions that do not require any input—that is, if we put enough money, we can transition to coke-giving states without any further input.

The primary goal of the present chapter is to provide you with the understanding of the basic mathematical notions that enable us to define finite automata formally. The secondary goal is to show you how how to read mathematical text, how to understand it, and how to write rigorous definitions.

In order to describe finite automata in a formal, mathematical way, we will need to specify the following parts:

- states,
- the starting state,
- final states,
- input symbols, and
- transition rules.

In the course of this chapter, we will introduce appropriate mathematical constructs for each part, together with basic explanations of these constructs.

This chapter is organized as follows. First, Section 2.1 introduces sets, which are the basic mathematical structures that we use throughout the whole chapter. Then, Section 2.2 briefly mentions a related concept to sets—sequences. Section 2.3 introduces so-called relations, which represent a formalization of rules of finite automata. After that, in Section 2.4, we take a look on functions, which are special relations that can be used to model deterministic versions of finite automata. Finally, the chapter is concluded by Section 2.5, where we put all the pieces together and obtain a formal definition of a finite automaton.

2.1 Sets

From the informal description of a finite automaton given in the introduction to the present chapter, we see that a finite automaton is composed of several objects. These objects are states, rules, and input symbols. From them, let us focus on states. First of all, they are labeled by a label by which we may distinguish them. Therefore, if you are given two states, you can immediately tell whether these two sets are, in fact, identical, or they are two distinct states. Furthermore, if you have a finite automaton, you may clearly say which states belong to the automaton and which states do not. This is the first important property. The second property is that the order of states does not matter. Indeed, the transition diagram can be drawn in many different ways and it does not make much sense of saying that one states is greater than some other state. When formalizing states, we would like to have a mathematical structure that can capture both of these properties. This brings us to sets.

What Is a Set?

A *set* is a collection of distinct objects[1].

Example 2.1. Examples of sets:

- Set V of three basic colors: $V = \{\text{red}, \text{green}, \text{blue}\}$.
- Set A containing four arrows: $A = \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$.

For each object, you can tell whether it belongs to the set or not, and that is the only property of sets. Therefore, sets are without any other structure than membership. If we have a set, we can only ask if an object is a member of (i.e. is present in) the set or not.

Note that this has the following two severe implications:

1. As the membership can be only true of false, an object cannot be contained in a set multiple times.
2. Objects in a set have no implicit ordering.

Example 2.2. A consequence of these implications is that the sets $\{1, 2, 3\}$, $\{1, 1, 2, 3\}$, and $\{2, 3, 1\}$ are all equal. In fact, these are just three different ways to describe a single set.

As you have already noticed, sets are conventionally denoted with capital letters, like P, Q, and R, or by other symbols, as we will see next. The members of a set are customarily called *elements* of the set. The fact that some object x is an element of a set P is written as $x \in P$; the converse situation, where x is not an element of P, is written as $x \notin P$.

[1] In fact, there are many definitions of sets in mathematics, and each definition have its pros and cons. However, in our case, this simple and intuitive definition (also called *naïve set theory*) will be satisfactory, as we will not use it for the problematic cases.

Example 2.3. Consider the set of all integers, which is usually denoted by $\mathbb{Z}$[2]. The objects -5, 0, 12456 are all elements of $\mathbb{Z}$ while the objects 3.25, `car`, and ♪ are not elements of this set. Using the notation utilizing $\in$ and $\notin$, we may write, for example, $-5 \in \mathbb{Z}$ and ♪ $\notin \mathbb{Z}$.

If you come from a programming background, you can think of sets as of arrays of boolean flags. Each object has its own flag in this array, and this flag says whether the object is a member of this set or not. This analogy reflects the boolean character of membership.

How To Describe a Set?

The first, obvious way to describe a set is to use a natural language to specify all the requirements imposed on the set.

Example 2.4. To define that $\mathbb{Z}$ is the set of all integers, we may write: "Let $\mathbb{Z}$ be the set of all integers." Or, as another example, we may say: "Let N be the set of states in Europe whose names do not start with F." The latter example, of course, assumes that we agree which states are part of Europe and which are not.

At first, the usual mathematical wording can sound a little bit strange: How can we say "Let N be the set of states in Europe."? What if the set of all states in Europe is A instead of N? Or, what if N is already some other set, like the set of all natural numbers? In this context, the word "let" is used more like in the sense "Let us choose N as a temporary name of the set of states in Europe.". The reason is that we need a short name for the set, so we do not have to write "the set of states in Europe" each time we want to say something about it.

The second way of describing a set is already very well known to you. It consists of enumerating all the elements of the set as a comma-separated list enclosed in curly brackets.

Example 2.5. To define a set Q consisting of numbers 1 through 5, we may write

$$Q = \{1, 2, 3, 4, 5\}$$

As another example, the set P of all playing card suits may be defined as

$$P = \{\clubsuit, \diamondsuit, \spadesuit, \heartsuit\}$$

[2] This symbol is called a *blackboard bold Z*, where *Z* stands for *Zahlen* (German for *numbers*).

As we have noted at the beginning of the chapter, the order in which the elements of a set are listed is irrelevant. As we will see later, when we need an ordered collection, we can use a *sequence* or *tuple* instead of a set.

Sometimes, there are just too many elements to list them all. Thus, when the meaning is clear, we may use three dots (the symbol "$\ldots$", also called an *ellipsis*) to abbreviate the definition.

Example 2.6. The set R consisting of numbers between 1 and 100 may be defined as

$$R = \{1, 2, \ldots, 100\}$$

As another example, $\{a, b, \ldots, z\}$ stands for all the lower-case letters of the English alphabet.

The last way of specifying a set is by giving a property that all its elements satisfy. This specification is of the form

$$Q = \{x : \text{some property that } x \text{ has to satisfy}\}$$

As usual, we first put the name of the set, then the equality sign, and after that the definition of the set. The definition is composed by giving the name of a variable (x in our case) that will be used in the specification of the property, then a delimiter (usually a colon), and after that, we prescribe the property.

Example 2.7. The set of all *natural numbers* $\mathbb{N}$ may be written as

$$\mathbb{N} = \{x : x \in \mathbb{Z}, x \geq 0\}$$

The comma in the above example reads as *and*. All in all, the above definition of $\mathbb{N}$ can be read in the following way: "The set $\mathbb{N}$ is defined as a set of all xs such that x is an element of $\mathbb{Z}$ and, at the same time, x is greater or equal to zero."

Sometimes, the symbol "|" is used instead of ":". Also, you may encounter a use of the logical operators $\vee$ (logical *or*) and $\wedge$ (logical *and*). The following definitions of $\mathbb{N}$ are equivalent to the definition from the previous example:

Example 2.8. The set of all natural numbers may also be written as

$$\mathbb{N} = \{x \mid x \in \mathbb{Z}, x \geq 0\}$$

Notice that we have used the symbol "|" instead of ":". Or, alternatively, by using the logical operator "$\wedge$", we may write

$$\mathbb{N} = \{x : x \in \mathbb{Z} \wedge x \geq 0\}$$

Or, simply, just by writing it down in words

$$\mathbb{N} = \big\{x \text{ such that } x \in \mathbb{Z} \text{ and } x \geq 0\big\}$$

All of the mentioned approaches may be combined together. Although mathematics use an artificial language that is more strict than a regular language, it is not so strict as a programming language. The most important rule for writing mathematics is that each sentence should have precisely one meaning and it should be exact and accurate.

What Relations Between Sets Are There?

Sets can be related to other sets in several ways. The most obvious one is called *equality*. As its name suggests, two sets are *equal* if they contain precisely the same elements. We write this by using the symbol "=", so, for example, if A and B are two equal sets, we write this as $A = B$. If two sets are not equal, we write this by using "$\neq$" instead of "=".

Example 2.9. Let M and N be two sets, defined as $M = \{1, 2, 3\}$ and

$$N = \big\{x : 1 \leq x \leq 3\big\}$$

Then, $M = N$. Notice that these two sets are equal even though they are defined in a different way. Indeed, as both definitions imply that the sets have the same elements, 1, 2, and 3, they are equal. In fact, we can think of N and M just as two different names given to the same set.

Next, let P be a set defined as $P = \{1, 2, 3, 4, 5\}$. Then, $N \neq P$, which also implies that $M \neq P$. Indeed, since M is just an another name of N and $N \neq P$, then M and P cannot have the same elements.

The second relation is containment. If every element of a set A is also an element of a set B, then we say that A is a *subset* of B, written as $A \subseteq B$.

Example 2.10. If $A = \{\clubsuit, \spadesuit\}$ and $B = \big\{\clubsuit, \diamondsuit, \spadesuit, \heartsuit\big\}$, then $A \subseteq B$.

An interesting property to note is that, when we have some set A, then it is always true that $A \subseteq A$. Even though it may seem strange, it actually fits the definition of a subset quite nicely. Indeed, as every element of A is also an element of A, $A \subseteq A$.

However, what if we want to distinguish the situation when the sets are equal from the situation where the second set contains also some other elements which are not in the first set? To satisfy such a need, we will now define a new relation called *proper subset*. In general, to introduce a new construct into a mathematical system, the notion of *formal definition* (or just *definition*) is usually used. A definition is just a piece of text exactly and unambiguously describing the behavior of our new construct. The wording of a definition should be chosen carefully, as the main objective is to have just one single way to interpret the text.

A definition is usually one of the most dense texts in mathematics. Even though a lot of times it is just a single sentence consisting of mathematical symbols, the amount of information encoded in this sentence can be written in a paragraph of regular text, or even more. Definitions should be approached as a piece of source code in a special "programming language" of mathematics—you have to think about every word and its meaning in the context of the mathematical structure you are working with.

So, let us formally define the relation of a proper subset:

Definition 2.11. Let P and Q be two sets. If $P \subseteq Q$ and $P \neq Q$, then we say that P is a *proper subset* of Q. We write this as $P \subset Q$.

Example 2.12. Consider the sets A and B from Example 2.10. We have seen that $A \subseteq B$. However, as $A \neq B$, we may also write $A \subset B$.

From the example above, we see that $A \subset B$ says something more than $A \subseteq B$. Indeed, the former says that (1) all elements of A are in B and (2) there are elements in B that are not in A. The latter says only (1).

If we have good definitions of the basic constructs, lots of other notions can be defined more easily by using such strong foundations. One example can be the definition of set equality. We can us the subset relation to define the equality of two sets in an another way:

Definition 2.13. Two sets P and Q are said to be *equal*, written as $P = Q$, if $P \subseteq Q$ and $Q \subseteq P$.

Let us stop now for a while and think about the above definition. If $P \subseteq Q$, then we know that all elements of P are in Q. Conversely, from $Q \subseteq P$, we know that every element of Q is in P. This, however, implies that both sets contain precisely the same elements, which makes them equal. Thus, by using well-defined constructs, our definition of equality has been shrunk and simplified from a paragraph of text into a single short sentence.

Are There Any Special Types of Sets?

We have already seen many sets, but we have not discussed any natural way to create a hierarchy from sets. Indeed, a set can be a member of other set[3].

Example 2.14. Let us consider the set of all card suits: $\{\clubsuit, \diamondsuit, \spadesuit, \heartsuit\}$. Naturally, we can have sets containing only the black suits, that is $B_1 = \{\clubsuit\}$, $B_2 = \{\spadesuit\}$, and $B_3 = \{\clubsuit, \spadesuit\}$. Finally, as these sets are also mathematical objects, we can put them together to form an another set:

$$A = \big\{\{\clubsuit\}, \{\spadesuit\}, \{\clubsuit, \spadesuit\}\big\}$$

However, please be careful—each of B_1, B_2, and B_3 are elements of A, but they are not subsets of A. Thus, you can write $B_1 \in A$, but you cannot write $B_2 \subseteq A$.

There exist a special, unique set, which is a subset of every other set—the *empty set*. This is the set containing no elements. Usually, it is denoted by $\emptyset$. Such a set may arise quite naturally, as can be seen in the following example.

Example 2.15. Define the set E by the following notation:

$$E = \big\{x : x \geq 0, x + 1 = x\big\}$$

The set E is composed of numbers that are greater or equal to zero and, at the same time, when we increment an element, we obtain the same element. After a little thinking, we come to a conclusion that there is no such number that satisfies both conditions. Therefore, E is, in fact, empty, which can be written as $E = \emptyset$.

The empty set has many interesting properties, some of which we mention next. First, it is the only set that contains no elements. Indeed, all sets that differ from the empty set have to contain at least one element. Second, the empty set is a subset of all sets.

We now move to another special type of a set. Consider a set, Q. Depending on the number of its elements, there may be many subsets of Q. An interesting set is the set of all subsets of Q, defined next.

Definition 2.16. Let Q be a set. The *power set* of Q, denoted by 2^Q, is the set of all subsets of Q, defined as

[3] Nevertheless, we have to be careful with this property because in naïve set theory, some paradoxes can arise. One of the most famous example is the so-called *Russell's paradox*: consider the set $A = \{x : x \notin x\}$. Having such set, we can ask whether $A \in A$. However, this question cannot be answered, as both possible answers lead to a logical contradiction. In order to circumvent this paradox, a lot of different set theories have been proposed. One of the most widely used theory is *Zermelo-Fraenkel set theory.*

$$2^Q = \{P : P \subseteq Q\}$$

Example 2.17. Consider the set $A = \{1, 2, 3\}$. Then, its power set is

$$2^A = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 3\}, \{1, 2, 3\}\}$$

An interesting fact to note is that for a set of n elements, its power set has precisely 2^n elements. Therefore, for a three-element set, like the one in the example above, its power set has eight elements.

A thought-provoking question is: What is the power set of the empty set? In symbols, what elements are in the set $2^{\emptyset}$? To answer this question, let us recall the definition of a power set. For a set Q, $2^Q = \{P : P \subseteq Q\}$. If we rewrite it by substituting $\emptyset$ for Q, we obtain

$$2^{\emptyset} = \{P : P \subseteq \emptyset\}$$

How many subsets does the empty set have? Exactly one: itself. Therefore, 2^Q is composed of precisely a single element: the empty set. In symbols,

$$2^{\emptyset} = \{\emptyset\}$$

It should be noted that $\{\emptyset\}$ is not the same as $\emptyset$. Indeed, $\{\emptyset\}$ is a set containing a single element whereas $\emptyset$ does not contain any elements. Do not intermix these two sets.

We now proceed to a classification of sets based on the number of their elements. Before we delve into details, let us consider the following example.

Example 2.18. Let A, B, and C be three sets, defined as $A = \emptyset$, $B = \{1, 2, 3, 4, 5\}$, and $C = \{x : x \in \mathbb{Z}, x \geq 0\}$. Since A is the empty set, it does not contain any elements, and so the number of its elements is 0. In B, there are five elements, 1 through 5. However, how many elements are in C? The answer is: infinitely many. Indeed, C is an example of a so-called *infinite* set, discussed next.

Definition 2.19. Let P be a set. If there is an integer n such that P contains precisely n elements, then P is a *finite set*; otherwise, P is an *infinite set.*

There are many examples of infinite sets in computer science—the set of all text files, set of all source code files for a given programming language, set of all valid network streams for given protocol, etc. Note that set of all source code files for some programming language is, in fact, a proper subset of the set of all text files.

If you think of sets as of binary arrays with membership flags, it can be somewhat difficult to imagine infinite arrays. However, to define an infinite set, you can create a computer program accepting some input data as a member of the set, and rejecting all other data. For example, to define the set of all valid Java programs, you can use the Java compiler—a compiler always tells you whether the given source code file is a valid Java program or not, and apparently, there is an infinite number of different Java programs that can be created[4].

For the definition of a finite automaton, we will use just finite sets (as a finite automaton has a finite number of states, input symbols and rules). However, infinite sets will be useful later for defining all the possible move sequences of a finite automaton.

What Operations Can Be Performed Over Sets?

There exist many operations that can be done over sets. In this text, we mention only the most commonly used ones.

In the first operation we are going to discuss, we take two existing sets and create a new set from them, which will contain elements of both of these sets. In other words, we unite the two sets into a single set.

Definition 2.20. Let P and Q be two sets. The *union* of P and Q, denoted by $P \cup Q$, is defined as

$$P \cup Q = \big\{x : x \in P \text{ or } x \in Q\big\}$$

The above definition reads as follows: "The union of P and Q is the set that contains all such x that are elements of P or Q." This implies that in the resulting set, we include elements of both P and Q, even if some object is an element of only one of these two sets. Furthermore, since every object can be in a set at most once, if both P and Q contain some same element, this element is included in $P \cup Q$ only once. The following example illustrates this operation on sets of card suits:

Example 2.21. Consider the sets $A = \{\clubsuit, \spadesuit\}$ and $B = \{\clubsuit, \diamondsuit\}$. Then,

- $A \cup B = \{\clubsuit, \spadesuit, \diamondsuit\}$;
- $A \cup A = A \cup \emptyset = \{\clubsuit, \spadesuit\}$.

Notice that the union of A with the empty set—as $\emptyset$ does not contain any item, it will not add anything to the union. If you think of sets as of boolean arrays, the union operation is, in fact, equivalent to the binary *or* operation on two binary arrays. In this

[4] One can argue that indeed there is an infinite number of programs, but the Java compiler have to run on a computer, and a computer has only finite memory. Thus, it can accept only finite number of programs up to some length. That can be true (at least until you upgrade your hardware), but anyway, the existence of infinity is rather a metaphysical question fitting more for philosophers than engineers. However, by actually thinking about an infinite inputs, an engineer can create much more elegant and efficient solutions that scale well to larger and larger architectures.

analogy, you can see that the resulting unified set cannot have two instances of a single object. Furthermore, during a union, there cannot arise any new object that is not in one of the two sets that are being united.

The second operation, called *intersection*, also constructs a new set from two existing sets. However, contrary to set union, it includes only the elements that are in both sets.

Definition 2.22. Let P and Q be two sets. The *intersection* of P and Q, denoted by $P \cap Q$, is defined as

$$P \cap Q = \{x : x \in P \text{ and } x \in Q\}$$

The above definition reads as follows: "The intersection of P and Q is the set that contains all such x that are elements of both P and Q." Therefore, if there is an element that is in only one of these two sets, it is not included into the resulting set. The following example illustrates this operation on sets of card suits:

Example 2.23. Let $A = \{\clubsuit, \spadesuit\}$, $B = \{\clubsuit, \diamondsuit\}$, and $C = \{\heartsuit, \diamondsuit\}$. Then,

- $A \cap B = \{\clubsuit\}$;
- $B \cap C = \{\diamondsuit\}$;
- $A \cap A = \{\clubsuit, \spadesuit\}$;
- $A \cap C = A \cap \emptyset = \emptyset$.

If you think of sets as of boolean arrays, the intersection operation is, in fact, equivalent to the binary *and* operation on two binary arrays.

The last operation we are going to discuss differs from both union and intersection—it is an operation over a single set rather than over two sets. Let P be a set. We might want to get objects which are *not* in P, which is precisely what the next operation does. However, to make this operation meaningful, we need to know what is the *universe* of all objects. The universe is usually a big superset of all the object we want to consider. If we are working with natural numbers, our universe will not contain negative numbers or fractions. On the other hand, if we are working with source codes of various programming languages, we would probably choose the set of all text files as our universe.

When we have our favourite universe chosen, we can define the *complement* operator as follows:

Definition 2.24. Let P a set, and $\mathbb{U}$ be the universe of all objects. Then, the *complement* of P, denoted by $\overline{P}$, is defined as

$$\overline{P} = \{x : x \in \mathbb{U}, x \notin P\}$$

The above definition reads as follows: "The complement of P is the set that contains all such x that are elements of $\mathbb{U}$, but, at the same time, are not elements of P." Therefore, if there is an element that is in the universe but not in P, it will be included in the resulting set. The following example illustrates this operation on sets of card suits:

Example 2.25. Set the universe to $\mathbb{U} = \{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\}$ and consider the set $A = \{\clubsuit, \diamondsuit\}$. Then,

- $\overline{A} = \{\heartsuit, \spadesuit\}$;
- $\overline{\emptyset} = \mathbb{U}$;
- $\overline{\mathbb{U}} = \emptyset$.

Once again, if you think of sets as of boolean arrays, the complement operation is equivalent to binary negation.

Towards Defining a Finite Automaton

Now, we know several things about sets, so we can try to define at least some parts of a finite automaton. Recall that a finite automaton consist of the following parts:

- states,
- the starting state,
- final states,
- input symbols, and
- transition rules.

As sets are, in fact, just collections of objects, we can create the set of all *states* (we will denote it by Q for future reference), set of *final states* (denoted by F), set of *input symbols* (denoted by Σ[5]) and set of *transition rules* (denoted by R). The *starting state* (denoted by s) will be just a designated element of the set of *states*—that is, $s \in Q$—and possibly also an element of the set of *final states*. Furthermore, the *final states* have to form a subset of all *states*, that is, $F \subseteq Q$. This covers the major part of our automaton, but two essential parts are missing:

1. The states and input symbols can be just simple objects, but the transition rules have to be more complex objects representing the relation between two states and one optional input symbol. Furthermore, there is initial start state and the ending state for each transition, so we cannot use the simple notion of an unordered set.
2. The automaton has to be defined as a complete object, not just a collection of some sets. Therefore, we could try to define it as a set $A = \{Q, \Sigma, R, s, F\}$. However, what if all the states are final states? That is, what if $F = Q$? Then, as a single object cannot be included in the set more than once, $\{Q, \Sigma, R, s, F\} = \{Q, \Sigma, R, s\} = \{\Sigma, R, s, F\}$, which can cause us some confusion and troubles. Furthermore, it would be practical to have some ordering on the components of a finite automaton so we know that the first component defines states, the second component defines input symbols, and so on.

[5] Σ is a Greek letter called *sigma* and it is usually used to denote the input alphabet in formal language theory.

Thus, in order to fully and conveniently define finite automata, we need to have another, more complex mathematical structure—a *sequence*.

2.2 Sequences

In the previous section, we have discussed sets, which are collections of objects, where their order does not matter and every object can be included at most once. However, sometimes it is handy to have a mathematical formalism in which we may specify order and include several instances of single object. This brings us to sequences.

A *sequence* is a list of objects. Contrary to a set, a sequence can contain an object more than once and the objects appear in a certain order. By analogy with a set, objects that appear in a sequence are called *elements* of the sequence. Elements in sequences are usually separated by a comma. Furthermore, to distinguish sequences from sets, we enclose all the elements of a sequence in the round brackets—symbols "(" and ")"—instead of curly brackets.

Example 2.26. A sequence of the numbers for five consecutive dice rolls:

$$R = (1, 3, 2, 6, 6)$$

A sequence of all the letters in the word *sequence*:

$$L = (s, e, q, u, e, n, c, e)$$

A different sequence containing the same letters, this time in an alphabetic order:

$$A = (c, e, e, e, n, q, s, u)$$

In the previous example, note the following important points:

1. The elements of a sequence are enclosed by round brackets. Thus, we can easily differentiate between sets and sequences.
2. A sequence can contain several occurrences of a single object—two instances of the number 6 in the first sequence, or three instances of the letter e in second and third sequence.
3. Even though L and A contain the same objects, $A \neq L$. In other words, the order of the objects in a sequence does matter.

As sets, sequences can be either *finite* or *infinite*. However, in this text, we will use only finite sequences. Infinite sequences are often used in terms of *mathematical series* [7].

Example 2.27. The Fibonacci sequence, where each member equals to the sum of its two previous members, is one of the most known infinite number sequence:

$$F = (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots)$$

Sequences can be infinite in both directions. For example, the sequence of even numbers ordered by value can be written as

$$E = (\dots, -4, -2, 0, 2, 4, 6, 8, 10, \dots)$$

Finite sequences are also called *tuples*. More specifically, sequences of two, three, four, five, six, and seven elements are called *pairs*, *triplets*, *quadruples*, *quintuples*, *sextuples*, and *septuples*, respectively.

Example 2.28. The coordinates of a point in an n-dimensional space can be represented as n-tuples. We can use pairs such as $(3, 5)$, $(-10, 12)$, or $(1.23, -4.8)$ for two dimensional points, or triplets as $(1, -2, 3)$ or $(99.76, 0.593, 2.67)$ for points in a three dimensional space.

Furthermore, sequences can contain other sequences in a similar fashion to sets.

Example 2.29. A sequence of the cards dealt during the blackjack game last night:

$$G = \big((6, \heartsuit), (3, \clubsuit), (4, \heartsuit), (9, \spadesuit)\big)$$

A sequence of the points you need to cross to draw a simple house in a single line:

$$P = \big((-1, -1), (-1, 1), (1, 1), (1, -1), (-1, 1), (0, 2), (1, 1), (-1, -1), (1, -1)\big)$$

Note that both of these examples consist of sequences of tuples.

Now that we have a little bit stronger tool to work with, we can continue in the formal definition of a finite automaton. Recall that we have defined the set of all states Q, set of inputs $\varSigma$, set of rules R, starting state s, and set of final states F. However, we had encountered the following two problems:

1. transition rules need to have start and end;
2. the automaton has to be defined as a complete object even if $F = Q$.

The second problem can be easily fixed by defining the finite automaton as a quintuple $M = (Q, \varSigma, R, s, F)$. Thus, even if $Q = F$, the automaton still have the same form of a quintuple with each component always on the same place.

The problem number one can be also solved by sequences. Each rule can be a triplet (p, a, q), where p is the initial state of the transition, a is some input, and q is the ending state of the transition. Furthermore, recall that some transitions do not have a corresponding input symbol. Therefore, we have to devise a new, special symbol denoting the “no-input” property of these transitions. In formal language theory, the symbol ε (a Greek letter called *epsilon*, used as an abbreviation for the word *empty*) is used in these kind of situations.

However, as we also want explore the various computation capabilities of finite automata, we should go even deeper down the rabbit hole of mathematical models. The transition rules are not only some ordinary sequences. In fact, they are very special sequences with constraints—both the starting and ending element of a rule need to be from the set of states, and the input element need to be from the set of inputs Σ (or ε). Furthermore, we want to explore the computation process of different input sequences by chaining the transitions one after another. However, not all the transitions can follow each other. Thus, we need to consider the relations between inputs and states in the transition rules. And, as you have already guessed, there is an “upgraded” version of a sequence, called *relation*, which we can use exactly for this purpose.

2.3 Relations

In mathematics and its applications, we often want to specify relationships between objects. For example, consider the *greater than* relation over integers. This relation is usually denoted by the symbol “>”. From primary school, we know that the numbers 9 and 2 (in this order) are related by this relation while 5 and 7 (in this order) are not related by >. This fact is usually written as $9 > 2$ and $5 \not> 7$. Notice that the order of the two numbers is important. Indeed, $9 > 2$, but $2 \not> 9$.

In this section, we are going to talk about such relations between objects. You will see how to define the term *relation* mathematically, how to specify your own relations, and how to use them.

Before we start with the definition of a relation, we will need the concept of the *Cartesian product* of two sets. The Cartesian product of two sets, P and Q, is the set of all pairs (x, y) such that x is an element of P and y is an element of Q.

Definition 2.30. The *Cartesian product* of two sets, P and Q, denoted by $P \times Q$, is defined as

$$P \times Q = \big\{(x, y) : x \in P \text{ and } y \in Q\big\}$$

In the above definition, the symbol “×” read as “times”. Note that the pairs (x, y) are, in fact, the special sequences of two objects called tuples (or, in this case, pairs).

Example 2.31. Consider the two sets $A = \{1, 2\}$ and $B = \{\clubsuit, \diamondsuit, \spadesuit, \heartsuit\}$. Then, their Cartesian product is

$$\begin{aligned} A \times B = \big\{&(1, \clubsuit), (1, \diamondsuit), (1, \spadesuit), (1, \heartsuit),\\ &(2, \clubsuit), (2, \diamondsuit), (2, \spadesuit), (2, \heartsuit)\big\}\end{aligned}$$

As you can see, we have included all possible combinations of 1 and 2 with $\clubsuit$, $\diamondsuit$, $\spadesuit$, and $\heartsuit$.

An interesting fact about the Cartesian product is that if both sets P and Q are finite (they have a finite number of elements), then their Cartesian product is also a finite set with mn elements, where m is the number of elements of P and n is the number of elements of Q.

Example 2.32. Consider the two sets A and B from Example 2.31 and their Cartesian product. The set A has two elements while B has four elements. Since both of these sets are finite, their Cartesian product should have $2 \cdot 4 = 8$ elements. And, by counting the number of elements of $A \times B$ in Example 2.31, we see that we have computed the number of elements correctly.

It is not necessary for the two sets in Definition 2.30 to be different. Indeed, we may compute the Cartesian product of a set with itself, like in the following example.

Example 2.33. Let $Q = \{1, 2, 3\}$ be a set. Then, the Cartesian product $Q \times Q$ has the following $3 \cdot 3 = 9$ elements:

$$Q \times Q = \big\{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)\big\}$$

It is also possible to do the Cartesian product of infinite sets or even a finite set with an infinite set, as we will see in the next two examples. When one of the sets is infinite, the resulting Cartesian product is also always infinite.

Example 2.34. Let P be a set defined as $P = \{+, -\}$. Then, the Cartesian product of P with $\mathbb{N}$ (the set of all natural numbers) is denoted by $P \times \mathbb{N}$ and contains an infinite number of pairs. Always, however, the first item in every pair is either $+$ or $-$ and the second one is a number from $\mathbb{N}$. Since $\mathbb{N}$ is an infinite set, the Cartesian product is also an infinite set.

Example 2.35. Consider the set of all natural numbers $\mathbb{N}$. The Cartesian product of $\mathbb{N}$ with itself—that is, $\mathbb{N} \times \mathbb{N}$—has an infinite number of pairs of numbers. If you take any two numbers i and j, then the pair (i, j) is in $\mathbb{N} \times \mathbb{N}$.

Now, after we have throughly explored the notion of the Cartesian product of two sets, we are ready to define mathematical relations formally. Informally, a relation is a set of pairs from the Cartesian product of two sets. That is, given two sets P and Q, a *relation* is a subset of their Cartesian product:

Definition 2.36. A *relation* R from a set P to a set Q is defined as

$$R \subseteq P \times Q$$

Example 2.37. Let us consider four persons: Jane, Paul, Don, and Elizabeth. Jane is 16 years old, Paul is 32 years old, and both Don and Elizabeth are 22 years old. We want to specify this age relation mathematically. To do this, we create a set

$$P = \big\{\text{Jane}, \text{Paul}, \text{Don}, \text{Elizabeth}\big\}$$

and define the relation $\mathit{age} \subseteq P \times \mathbb{N}$ as follows:

$$\mathit{age} = \big\{(\text{Jane}, 16), (\text{Paul}, 32), (\text{Don}, 22), (\text{Elizabeth}, 22)\big\}$$

In this way, we have formally specified the relation between the four persons and their age.

The fact that two objects x and y are in a relation R is denoted by $(x, y) \in R$. Nevertheless, to improve readability, it is common to use the infix notation xRy. Indeed, you have probably almost never seen the notation $(6, 3) \in >$; rather, what you have seen is $6 > 3$, which is of the infix form.

Since a relation is a set, all common properties and operations over sets apply to relations as well. As a specific example, if a relation has a finite number of elements, it is a *finite relation*; otherwise, it is an *infinite relation*.

Example 2.38. The relation from Example 2.37 is finite while the mathematical relation $>$ over integers is infinite.

In the case we have the relation over the Cartesian product of a single set—that is, over $A \times A$—instead of saying "relation from A to A", we may simply say "relation over A".

Sometimes, we need to relate more than two objects. Therefore, it would be handy to extend the relation to more sets. Let us begin by extending the definition of Cartesian product to n sets as follows:

Definition 2.39. The *Cartesian product* of n sets, P_1 through P_n, where $n \geq 1$, is denoted by

$$P_1 \times P_2 \times \cdots \times P_n$$

and defined as

$$P_1 \times P_2 \times \cdots \times P_n = \big\{(x_1, x_2, \ldots, x_n) : x_1 \in P_1, x_2 \in P_2, \ldots, x_n \in P_n\big\}$$

Now, we can define an n-ary relation as a subset of the above-defined n-ary Cartesian product.

Definition 2.40. For $n \geq 1$, an n-ary *relation* R over sets $P_1, P_2, \ldots, P_n$ is defined as

$$R \subseteq P_1 \times P_2 \times \cdots \times P_n$$

For relations between two objects, we use the term *binary relation*. When we are dealing with three components, we may use the term *ternary relation*. Ternary relations will be useful for defining the transition rules of a finite automaton, as these rules are composed of three objects: initial state, input symbol, and the ending state. Therefore, we will define the set of transition rules as a subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$.

However, before we delve into the domain of finite automata, let us introduce one more special type of a relation called *function*. The reason is that functions are, in fact, one of the hottest candidate for the most important concept in mathematics. Furthermore, as you will see, they will be useful for defining a special variant of a finite automaton.

2.4 Functions

Functions are just special relations with two restrictions. The main idea of a function is that one object can be related just to a single another object. This can be convenient in many situations. For example, consider the *age* relation from Example 2.37. From a purely mathematical viewpoint, the following relation, age' (read "*age* prime"), is valid:

$$age' = \big\{(\text{Jane}, 16), (\text{Jane}, 20), (\text{Don}, 22)\big\}$$

In this relation, Jane is associated with the ages 16 and 20, and Paul and Elizabeth are not associated to any age. In our understanding of the notion of an age, a person's age is unique and all people have some age. From this viewpoint, the age' relation is not valid, even though it is mathematically correct. What we want is to specify that for two sets, P and Q, the relation $R \subseteq P \times Q$ should satisfy the following two properties:

(1) *Uniqueness.* For every $x \in P$, there is at most one $y \in Q$ such that $(x, y) \in R$. Recall that xRy means that $(x, y) \in R$.
(2) *Totality.* For every $x \in P$, there exists $y \in Q$ such that $(x, y) \in R$.

This brings us to functions, as these two properties are exactly the restrictions we need to impose on a relation in order to call it a function.

Definition 2.41. Let P and Q be two sets. A *function* f from P to Q, is a relation from P to Q such that for every $x \in P$, there is precisely one $y \in Q$ such that $(x, y) \in f$.

Based on the above definition, we see that a function is a special type of a relation. Therefore, all functions are relations, but some relations are not functions (as we have seen in the introduction to this section).

Example 2.42. Consider the relation *age* from Example 2.37 over the sets

$$P = \{\text{Jane}, \text{Paul}, \text{Don}, \text{Elizabeth}\}$$

and the set of all natural numbers $\mathbb{N}$:

$$\mathit{age} = \{(\text{Jane}, 16), (\text{Paul}, 32), (\text{Don}, 22), (\text{Elizabeth}, 22)\}$$

Clearly, as each name have *exactly* one corresponding number, it is a function (note that even though each name has to have exactly one value assigned to it, a single value can be assigned to multiple names, as in the case of the value 22, which is assigned to both Don and Elizabeth).

On the other hand, if you consider the relation

$$\mathit{age}_2 = \{(\text{Jane}, 16), (\text{Jane}, 20), (\text{Paul}, 32), (\text{Don}, 22), (\text{Elizabeth}, 22)\}$$

you can see that as Jane has two corresponding values, 16 and 20, attached to it, it is not a function. Likewise, the relation

$$\mathit{age}_3 = \{(\text{Jane}, 16), (\text{Paul}, 32), (\text{Elizabeth}, 22)\}$$

is not a function over $P \times \mathbb{N}$ because even though it does not have multiple values attached to some name, it is not total. Indeed, Don has no corresponding value attached to itself.

In terms of functions, instead of $(x, y) \in f$, we usually write $f(x) = y$. This means that the function f assigns y to x. The reason why we may simply write $f(x) = y$ instead of $(x, y) \in f$ follows from the uniqueness and totality properties of functions.

Example 2.43. Consider for one more time the relation *age* from Example 2.37. By using the new notation, we can define *age* in the following way:

$$\mathit{age}(\text{Jane}) = 16,\ \mathit{age}(\text{Paul}) = 32,\ \mathit{age}(\text{Don}) = 22,\ \mathit{age}(\text{Elizabeth}) = 22$$

Now we know all mathematical tools that we need in order to fully formalize the concept of a finite automaton and its computation process.

2.5 Finite Automaton

In this section, we will put together everything we have learned in the previous four sections concerning sets, relations, sequences, and functions. Indeed, finally, we will present a complete formal definition of a finite automaton.

For the sake of compactness, we will review our previous argumentations and discussions about what do we need and what mathematical concepts we should use to model certain features.

- *States.* As its name suggests, a finite automaton has a finite number of states. As states are not required to be ordered and a single state may appear only once in the automaton, an appropriate model is a set. Furthermore, as there has to be only a finite number of states, this set has to be finite.
- *Inputs.* Another component of a finite automaton to be specified are inputs. We will model them by analogy with states—that is, we use a finite set.
- *Transition rules.* A finite automaton has a finite number of transition rules, which take it from a state by optionally reading an input symbol into another state. Therefore, we see that states and input symbols are in a relation, which will be a suitable mathematical concept do model rules. Indeed, we may use a relation which relates two states with an input symbol. Furthermore, we have to use the unique non-input symbol ε in rules with no input symbol.
- *Start state.* Every finite automaton has a specified start state, from which it starts its computation. The start state will be a designated element of the set of states.
- *Final states.* There also have to be some designated states in which the automaton can finish its computation. We will describe them as a subset of the set of states.

Based on the above observations, we see that a finite automaton should have five components. To put all of them together, we will use a sequence—more specifically, a quintuple (recall that a quintuple is a sequence having five elements).

Now, we are ready to define a finite automaton formally.

Definition 2.44. A *finite automaton* is a quintuple

$$M = \big(Q, \Sigma, R, s, F\big)$$

where

- Q is a finite set of *states*,
- Σ is a finite set of *input symbols*,
- $R \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is a finite relation, called the set of *rules*,
- $s \in Q$ is the *start state*, and
- $F \subseteq Q$ is a set of *final states.*

Let us go over the definition and make several notes. First of all, we have defined a finite automaton to be a quintuple rather that a five-element set. Recall that the reason behind this choice is twofold:

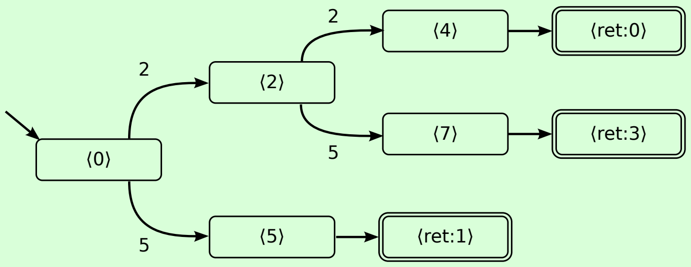
1. In a sequence, the meaning of the components is prescribed by their order. Therefore, when we simultaneously discuss two finite automata, we may write just $M = (Q, \Sigma, R, s, F)$ and $H = (O, T, P, p, G)$ without any need to say that O is the set of states of H, T is the set of inputs of H, etc.
2. A quintuple stays a quintuple even if several elements are the same. For example, when a finite automaton has $Q = F$, which means that all its states are final, such a finite automaton is still a quintuple. If we defined it as a five-object set, we would end up with a four-object set because in a set, there cannot be more than one occurrence of the same object. In our case, the same objects are Q and F.

The first and second component are a finite set of states and a finite set of input symbols. The necessity to have both of these sets finite stems from the nature of a finite automaton.

Rules are modelled by a relation $R \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$. First, note that we are using the ε symbol in addition to input symbols. Second, members of R are referred to as *rules*, and instead of $(p, a, q) \in R$, we will write $pa \to q$. The reason behind this different syntax is that it makes the meaning of the rule more clear and convenient. A rule $pa \to q$ denotes the fact that when the automaton is in the state p and the current input is a, it can move to the state q. Of course, you may write (p, a, q), but the simplified notation is more clear. Furthermore, a may be ε, meaning that this rule is applicable irrespective of the current input symbol. In such a case, we may write just $p \to r$.

Finally, the start state is a designed state of Q, and the set of final states is some subset of Q.

Example 2.45. Recall the model of a coke-vending machine from Figure 2.1 on page 7, which is for convenience displayed below.



Armed with the definition of a finite automaton, we will now define the underlying finite automaton behind this machine formally. Let $M = (Q, \Sigma, R, \langle 0\rangle, F)$ be a finite automaton, where

$$
\begin{aligned}
Q &= \{\langle 0\rangle, \langle 2\rangle, \langle 4\rangle, \langle 5\rangle, \langle 7\rangle, \langle \text{ret:}0\rangle, \langle \text{ret:}1\rangle, \langle \text{ret:}3\rangle\} \\
\Sigma &= \{2, 5\} \\
R &= \{\langle 0\rangle 2 \to \langle 2\rangle, \langle 0\rangle 5 \to \langle 5\rangle, \langle 2\rangle 2 \to \langle 4\rangle, \langle 2\rangle 5 \to \langle 7\rangle, \\
&\quad\ \langle 4\rangle\varepsilon \to \langle \text{ret:}0\rangle, \langle 7\rangle\varepsilon \to \langle \text{ret:}3\rangle, \langle 5\rangle\varepsilon \to \langle \text{ret:}1\rangle\} \\
F &= \{\langle \text{ret:}0\rangle, \langle \text{ret:}1\rangle, \langle \text{ret:}3\rangle\}
\end{aligned}
$$

Note that we use the angle brackets “⟨” and “⟩” to distinguish between states and inputs in order to improve readability.

We close this chapter by discussing a variant of a finite automaton. From a practical viewpoint, a general finite automaton has the following two disadvantages, both stemming from its non-deterministic nature:

(1) When there are two rules $pa \to q_1$ and $pa \to q_2$ with distinct q_1 and q_2, by reading a, which of these rules should the automaton choose?
(2) When there are two rules $pa \to q_1$ and $p\varepsilon \to q_2$ and the current input symbol is a, should the automaton read a and move to q_1 by using the first rule or not reading anything and move to q_2 by using the second rule?

To make finite automata more applicable in practice, we would like them to be *deterministic*. That is, we do not want any ε-rules, which are rules of the form $p\varepsilon \to q$, and from a state, with every input symbol, there should be at most one rule which can take us to the next state by reading the input symbol. This brings us to the following definition of a deterministic finite automaton, which satisfy the properties (1) and (2) above.

Definition 2.46. Let $M = Q, \Sigma, R, s, F)$ be a finite automaton. M is a *deterministic finite automaton* if R is a function from $Q \times \Sigma$ to Q.

That is it. By utilizing the notion of a function, we were able to create a succinct definition of a deterministic finite automaton. For each combination of state and input symbol, we know exactly whether the automaton will stop, or what will be the next configuration.

Example 2.47. Consider the following finite automaton:

$$M = \big(Q, \Sigma, R, s, F\big)$$

where $Q = \{s, f\}$, $\Sigma = \{a, b, c\}$, $R = \{sa \to s, sb \to f, fc \to f\}$, and $F = \{f\}$. Note that for each rule, the combination of a state and input has at most one resulting state, and there are no ε-rules. Thus, M is a deterministic finite automaton.

3

Defining Computation and Accepted Language

In the previous chapter, we have successfully defined the notion of a finite automaton. To do this, recall that we had to explain the basics behind the underlying mathematical concepts, such as sets, sequences, relations, and functions. In this chapter, we continue with the investigation of finite automata by defining their computation and accepted language. We begin by giving you a rough idea about these terms.

As you may know, a finite automaton works as follows. It starts from the start state and, by reading the inputs from the environment, it uses its rules to change states. Once it appears in a final state and there are no more input symbols, we say that it *successfully finishes its computation*. Therefore, the term *computation* means reading input symbols from the environment and accordingly changing states.

The environment may produce many different sequences of symbols. In fact, such sequences may be composed of an arbitrary number of characters, all of which are from the set of input symbols. If we only consider sequences of length 5 and there are 2 different input symbols, by using high-school mathematics, we may compute that there exist $2^5 = 32$ different sequences of length 5. In formal language theory, sequences of symbols from the environment are usually represented by *strings*, which are, in fact, just finite sequences. As you may recall from your formal language theory class, if we talk about strings, then instead of (a, b, c, d), we write just $abcd$—that is, we completely leave out the brackets and commas. In the remainder of this text, the inputs of a finite automaton are always assumed to be given in the form of strings. On the other hand, in some applications—for example, when using a finite automaton to analyze packets from a network stream—you do not know the complete input string beforehand. However, its not a problem for our model because a finite automaton reads the input sequentially, one symbol at a time, and it does not do any jumps.

Generally, a finite automaton may be given a large variety of input strings. With some of them, upon reading them all, it may enter a final state. However, with the rest of them, it either ends up in a non-final state or may not even be able to read them completely (it may get “stuck”). We are particularly interested in strings that will lead the automaton from the start state into a final states. Indeed, the set of all such strings will form the *accepted language*.

The present chapter is organized as follows. First, in Section 3.1, we formalize the notion of a computation performed by a finite automaton. Then, based on this formalization, we define the language accepted by a finite automaton. Finally, Section 3.3 discusses so-called closures, which underly the definition a computation.

3.1 Computation

We begin by defining a computation. Recall that from the informal description given in the introduction to this chapter, it is a process of reading input symbols and changing states. Consider a finite automaton M in its start state, and a string w made of input symbols. Then, if there is a rule $sa \to p$, where a is the leftmost symbol of w (or the empty string) and p is some state, then M may read a and go to p. Thus, it ends in p and the remaining input string is x, where $w = ax$; informally, x is the unread part of the string. Now, what if someone asks us to give an *instantaneous description* of the computation process? An instantaneous description is information that fully defines the current state of computation. Of course, the instantaneous description of M at the beginning of the current computation is s with w because M is in the start state and nothing has been read. However, what about the situation after a has been read? Well, we can remember all the already accessed states (in our case, s and p), all the already read symbols (a in our case), and the unread part of the input string (x in our case). However, notice that the information about the already accessed states and the read symbols is of no use in further computation. Indeed, what matters is the current state and the unread part of the string because old states and read symbols cannot affect the rest of the computation in any way. So, after a has been read, it suffices to remember just the current state, p, and the unread part of the input string, x. This brings us to the notion of a configuration.

Before giving a formal definition of a configuration, we first try to see what we can use to define it. Consider a finite automaton, $M = (Q, \Sigma, R, s, F)$. States are taken from the set of states Q. From what state do we take strings? The set of inputs Σ does not suffice because it contains only symbols so no string longer than a single symbol can be created in this way. If you remember the basics of formal language theory, you should know the answer: from Σ^*. Recall that Σ^* denotes the set of all strings over the symbols in Σ. So, to define a configuration, we may use the sets Q and Σ^*. There are two ways of joining these two sets together, (1) and (2), discussed next.

(1) As a configuration, we may consider a pair (p, x), where $p \in Q$ is the current state and $x \in \Sigma^*$ is the unread part of the input string. Therefore, the set of all configurations will be $Q \times \Sigma^*$.
(2) A configuration will be a single string of the form px, where p and x are defined as in the above case. Then, the set of all configurations will be $Q\Sigma^*$ (a concatenation of Q with Σ^*).

Both of these two approaches are used in books. We have chosen to follow the first approach, so a configuration will be a pair, not a single string. The reason for this choice is somewhat better readability of the text. However, it should be noted that both approaches are equivalent.

We proceed to a formal definition of a configuration.

Definition 3.1. Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton. A *configuration* of M is any pair from $Q \times \Sigma^*$.

Example 3.2. Consider the finite automaton M from Example 2.47, whose definition is for convenience repeated here—that is,

$$M = \big(Q, \Sigma, R, s, F\big)$$

where $Q = \{s, f\}$, $\Sigma = \{a, b, c\}$, $R = \{sa \to s, sb \to f, fc \to f\}$, and $F = \{f\}$. Then, examples of configurations are (s, abc), $(p, ccbd)$, and (f, ε). Notice that the unread part of the input string can be empty, like in the case of (f, ε).

Alright, so now we know what is a configuration and we can talk about a computation. Basically, a computation is a process of movement from one configuration into another one. How to make a single move? By using a rule of the automaton. For example, let us say that we are in a configuration (p, ax), where p is the current state and ax is the unread part of the input, and $pa \to q$ be a rule which, by reading a, ends up in the state q. Therefore, by using this rule, we may move from (p, ax) into the configuration (q, x). Thus, in order to explore the possible computations of a finite automaton, we would like to connect these two configurations together. As one of the best suited mathematical tool for connecting two object is a relation, we use it for this purpose. This connection is called a *move* and it is defined next.

Definition 3.3. Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton. The *direct move relation* over $Q \times \Sigma^*$, symbolically denoted by $\vdash_M$, is defined as follows: $(p, ax) \vdash_M (q, x)$ in M if and only if $pa \to q \in R$ and $x \in \Sigma^*$.

Let us examine the definition more thoroughly. The first thing to realize is that the move relation depends on the specific automaton we are exploring. For the purpose of the definition, we will call this automaton M and we will also name its components.

Then, we chose the name and the symbol for the relation—in our case, it is a "direct move relation" and "$\vdash_M$" respectively—as well as specify the set over which the relation is defined—in our case, it is the set of all configurations. Observe that the naming of the automaton's components in the first part of the definition will allow us to conveniently denote the set of all configurations simply as $Q \times \Sigma^*$. Recall that if we say that a relation is over $Q \times \Sigma^*$, it means that the relation is from $Q \times \Sigma^*$ to $Q \times \Sigma^*$—that is, $\vdash_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$.

Finally, we need to choose which pair will be in the relation, and which pair will be not. For this, we want to use the rules of the automaton—we want to connect only such configurations for which there is a valid transition rule. Thus, (p, ax) will be connected with (q, x) if and only if there is a transition from the state p to the state q with a as the input symbols—$pa \to q \in R$. If we would end the definition right there, there could be a confusion over x—is x meant to denote a symbol or a string? As we need the definition to have only one possible interpretation, we need to specify that x can be any string of input symbols, thus $x \in \Sigma^*$.

Note that there are many ways of defining this relation. An alternative definition (with the same meaning) using shorter wording is presented below:

Definition 3.4. Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton. The *direct move relation* over $(Q \times \Sigma^*)$, symbolically denoted by $\vdash_M$, is defined as follows:

$$\vdash_M = \big\{\big((p, ax), (q, x)\big) : pa \to q \in R, x \in \Sigma^*\big\}$$

Examples of some direct moves follows.

Example 3.5. Consider the finite automaton M from Example 3.2. Then, for example, $(s, abc) \vdash_M (s, bc)$ by using the rule $sa \to s$, $(s, bc) \vdash_M (f, c)$ by the rule $sb \to f$, and $(f, c) \vdash_M (f, \varepsilon)$ by using the rule $fc \to f$.

Note that these moves can in fact follow one another. This brings us to the computation process of a finite automaton. Basically, a computation is a sequence of direct moves. So, when we start from a configuration (p, x), perform several moves by using the direct move relation, and end up in a configuration (q, y), then this is a computation.

If we start in a configuration c_1 and we know that $c_1 \vdash_M c_2$, $c_2 \vdash_M c_3$, and $c_3 \vdash_M c_4$, we can simply chain the adjacent moves together by writing $c_1 \vdash_M c_2 \vdash_M c_3 \vdash_M c_4$. As you can see, it would be convenient for us to describe all the possible configurations that can be reached from c_1. Moreover, we want to include c_1 itself, as when the automaton is in c_1, c_1 is already "reached".

In order to do so effectively, we can extend our direct move relation for exactly this purpose. Thus, we will create a new relation based on the old one, where we will add all the successors of a configuration to the new relation (not just the direct, adjacent one). A formal definition of this idea follows.

Definition 3.6. Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton. If

$$c_1 \vdash_M c_2 \vdash_M \cdots \vdash_M c_n$$

where c_i is a configuration of M for $1 \leq i \leq n$ and $n \geq 1$, then we write $c_1 \vdash_M^* c_n$ (a *computation*).

Observe how we have extended the old relation to form the new relation with a similar symbol ($\vdash_M^*$). Also, note that each c_i can be any valid configuration—that is, $c_i \in Q \times \Sigma^*$—as long as there is a valid direct move between them.

The last thing to note is that by defining $n \geq 1$, we are also including the special case of $n = 1$. Let us substitute this special case in the definition of $c_1 \vdash_M^* c_n$, resulting in $c_1 \vdash_M^* c_1$. Thus, we have defined the relation of the configuration with itself, which is exactly what we wanted—c_1 can be indeed reached from c_1, as it is already reached without making any move.

You will see another, much more simpler definition of a computation in Section 3.3, where we will discuss so-called closures. Until then, we will use the above definition.

Example 3.7. Consider the finite automaton M from Example 3.2. Then, for example,

$$(s, abc) \vdash_M^* (f, \varepsilon)$$

by using the rules $sa \to s$, $sb \to f$, and $fc \to f$, but also

$$(s, abc) \vdash_M^* (s, abc)$$

by using no rules.

With a full understanding of what a computation means, we may proceed to the definition of a language accepted by a finite automaton.

3.2 Accepted Language

As we have said earlier, we are particularly interested in the strings of input symbols that bring the automaton to a final state. A set of all such strings is called the *accepted language* of the automaton, and we are now going to define this notion formally.

Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton. We already know that the start configuration is of the form (s, w), where w is the input string from Σ^*. Then, to read w and end up in a final state, M has to use its rules. Therefore, it has to pass several configurations until it reaches a configuration of the form (f, ε), where f is a final state from F. As we have seen in the previous section, such a pass over configurations can be expressed by using $\vdash_M^*$. Based on these observations, we may define the accepted language in the following way.

Definition 3.8. Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton. The *accepted language* by M is denoted by $L(M)$ and defined as

$$L(M) = \big\{w : w \in \Sigma^*, (s, w) \vdash_M^* (f, \varepsilon) \text{ with } f \in F\big\}$$

The above definition reads as follows: "The accepted language of a finite automaton M is denoted by $L(M)$, which is the set of all w from Σ^* that satisfy the condition that from the starting configuration (s, w), M can make a sequence of moves ending in some final state f with all of the input being completely read."

Example 3.9. Consider the finite automaton M from Example 3.2. By inspecting its set of rules and the way they can be used during a computation, we immediately see that the accepted language is composed of strings which start with an arbitrary number of as, followed by a single occurrence of b, and ended by an arbitrary number of cs. By using the standard notion involving operations over languages, we may state that the language accepted by M is

$$L(M) = \{a\}^*\{b\}\{c\}^*$$

We are going to move to the final section of this chapter, where we see a simpler definition of a computation that is based on so-called *relational closures*.

3.3 Closures

In Section 2.3, we have discussed relations, which allow us to specify relationships between mathematical objects. Apart from the basics behind relations, there exist many special types of relations. We have seen one example of a special type of a relation in Section 2.4, called functions. In this section, we will see another two special types of relations. Our goal is to define computation of a finite automaton in a more concise way than provided by Definition 3.6.

Consider a set, Q, and a relation over Q, $R \subseteq Q \times Q$. In general, there may be some elements which are related to itself, and there may be elements which are not related to itself. In symbols, there may exists $a \in Q$ such that aRa but there also may be some $b \in Q$ such that bRb does not hold. The first special type of a relation is a relation in which every element is related to itself.

Definition 3.10. Let Q be a set and $R \subseteq Q \times Q$ be a relation over Q. If aRa for every $a \in R$, then R is a *reflexive* relation.

Example 3.11. Consider the standard relation $\geq$ on integers. Obviously, if we take any integer i, then $i \geq i$. For example, $6 \geq 6$. Therefore, we see that the relation $\geq$ is reflexive.

As an example of a relation that is not reflexive, consider the relation $>$ on integers. Indeed, $i \not> i$ for every integer i. For example, $6 \not> 6$.

Considering finite automata, is the direct move relation reflexive? The next example gives the answer.

Example 3.12. Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton. We were asked whether the relation of a direct move $\vdash_M$ is reflexive or not. How should we approach this question?

Well, by looking at the definition of a reflexive relation. It says that in a reflexive relation, every element is related to itself. What are the elements of $\vdash_M$? Configurations. Therefore, consider any configuration (p, w), where $p \in Q$ and $w \in \Sigma^*$. Does it hold that $(p, w) \vdash_M (p, w)$? Well, that depends whether R contains the rule $p\varepsilon \to p$. If so, then $(p, w) \vdash_M (p, w)$ holds; otherwise, it does not hold. Therefore, only automata with the rule $p\varepsilon \to p$ for *every* state $p \in Q$ have reflexive direct move relation. Thus, generally, the relation of a direct move can be reflexive, but not in all cases.

The second special type of a relation will be required to satisfy a different property, called the *transitive property*. This is the requirement that whenever an element a is related to an element b, and b is in turn related to an element c, then a is also related to c.

Definition 3.13. Let Q be a set and $R \subseteq Q \times Q$ be a relation over Q. If every $a, b, c \in Q$ satisfy that if aRb and bRc imply that aRc, then R is a *transitive* relation.

Example 3.14. Once again, consider the standard relation $\geq$ on integers. We have seen that it is a reflexive relation. Is it also transitive? If it is, then for every integers i, j, k, if $i \geq j$ and $j \geq k$, then $i \geq k$. This is true. For example, $6 \geq 4$ and $4 \geq 3$, and $6 \geq 3$.

What is an example of a relation that is not transitive? Consider the set of persons

$$P = \{\text{Diana}, \text{Sarah}, \text{Elinor}\}$$

and the relation

$$\mathit{mother} = \{(\text{Diana}, \text{Sarah}), (\text{Sarah}, \text{Elinor})\}$$

The meaning of this relation is that Diana is the mother of Sarah and Sarah is the mother of Elinor. This relation is clearly not transitive because even though Diana is the mother of Sarah and Sarah is the mother of Elinor, it is not true that Diana is the mother of Elinor.

By analogy with the question whether the direct move relation of finite automata is reflexive, we will now check whether it is transitive.

Example 3.15. Consider any finite automaton $M = (Q, \Sigma, R, s, F)$. The question is: is the direct move relation $\vdash_M$ transitive? To answer the question, recall what a relation has to satisfy to be transitive. Have you recalled that? If so, then consider $(p, x) \vdash_M (q, y)$ and $(q, x) \vdash_M (r, z)$, where $p, q, r \in Q$ and $x, y, z \in \Sigma^*$. Does $(p, x) \vdash_M (r, z)$ hold? Well, just like in Example 3.12, it depends on whether there is a rule which takes M from (p, x) into (r, z). Therefore, we see that the direct move relation is generally not transitive.

Therefore, we see that, generally, the direct move relation is neither reflexive nor transitive. A question you may now ask is that if we are given a relation, that is not reflexive and transitive, can we make such a relation reflexive and transitive? The answer is yes, by computing the so-called *reflexive and transitive closure* of the relation. Informally, what we will do is that we are going to add elements to the original relation until it is both reflexive and transitive.

Definition 3.16. Let Q be a set and $R \subseteq Q \times Q$ be a relation over Q. The *reflexive and transitive closure* of R is denoted by R^* and it is the relation with the following three properties:

(i) *Reflexivity and transitivity*: R^* is both reflexive and transitive.
(ii) *Containment*: $R \subseteq R^*$
(iii) *Minimality*: There is no other reflexive and transitive relation relation R_2 such that $R_2 \subset R^*$ satisfies (i) and (ii).

Let us go over this slightly more complicated definition. We begin with a relation $R \subseteq Q \times Q$ with no imposed requirements. Its transitive and reflexive closure, R^*, has to satisfy the three properties (i) through (iii). The first property says that the closure has to be both reflexive and transitive. The second property requires R^* to contain all the elements of R. Of course, there may more elements. The third condition then requires that R^* is the minimal relation, in the sense that there is no other transitive and reflexive relation R_2 that is strictly smaller than R^* and satisfies the first two conditions. The third condition ensures that we add just the items we really need, and not add some items which are not necessary.

Example 3.17. Consider the following set of cities

$$C = \{\text{Prague}, \text{Vienna}, \text{New York}, \text{Ottawa}\}$$

and a relation that says you can get from a city to another one by taking a direct flight:

$$F = \{(\text{Prague}, \text{Vienna}), (\text{Vienna}, \text{New York}), (\text{New York}, \text{Ottawa})\}$$

For simplicity, we assume that there is no way back—that is, even though you can fly from Prague to Vienna, there is no direct flight from Vienna back to Prague.

In what follows, we will now construct a reflexive and transitive closure of F, which will be denoted by F^*. While F means that you can take a *single* flight from a city to another city, F^* means that you can get from a city to another city by taking *as many flights as needed.* This is the general meaning of the reflexive and transitive closure.

Alright, so what should we do? First, to satisfy (ii), we include all elements of F into F^*:

$$F^* = F$$

Therefore, we have covered the possibility of getting into a city by taking a single direct flight. The next step is to make F^* reflexive. To this end, we extend it in the following way:

$$F^* = F^* \cup \big\{(\text{Prague}, \text{Prague}), (\text{Vienna}, \text{Vienna}),\\ (\text{New York}, \text{New York}), (\text{Ottawa}, \text{Ottawa})\big\}$$

Even though it may seems strange, this extension of F^* says that when your are in a city X, then you do not have to take *any* flights to get into X. This makes sense, right? Now, F^* is reflexive and satisfies (ii) and a half of (i).

To complete the second half of (i), we have to make F^* transitive:

$$F^* = F^* \cup \big\{(\text{Prague}, \text{New York}), (\text{Prague}, \text{Ottawa}), (\text{Vienna}, \text{Ottawa})\big\}$$

After this extension, we see, for example, that we may fly from Prague into Ottawa—that is,

$$\text{Prague}\, F^*\, \text{Ottawa}$$

Observe that Prague F Ottawa does not hold because there is no direct flight from Prague to Ottawa.

We have promised to give you a very concise and simple definition of computation—that is, the definition of $\vdash_M^*$ for a finite automaton M. We keep our promise. As its denotation suggests, $\vdash_M^*$ is simply the transitive and reflexive closure of $\vdash_M$. Indeed, if you recall the previous definition of $\vdash_M^*$, for convenience displayed below, you can see it is similar to the definition of the reflexive and transitive closure:

Definition 3.18. Let $M = (Q, \Sigma, R, s, F)$ be a finite automaton. If

$$c_1 \vdash_M c_2 \vdash_M \cdots \vdash_M c_n$$

where c_i is a configuration of M for $1 \leq i \leq n$ and $n \geq 1$, then we write $c_1 \vdash_M^* c_n$ (a *computation*).

The transitive property follows from the chain of configurations c_1 to c_n. The reflexive property follows from the special case of $n = 1$, where, by substituing $n = 1$ to $c_1 \vdash_M^* c_n$, we get $c_1 \vdash_M^* c_1$.

4 Mathematical Statements and Their Proofs

Throughout your studies, you have almost certainly encountered mathematical sentences called theorems, followed by a reasoning showing that what is written in the theorem is true. A usual reaction of a student is slight confusion, but in some (and we hope rare) cases, it can be even immense fear, followed by depression. People fear what they do not know. To help you overcome this confusion and potential fear, this four-section chapter explains the basics of mathematical statements and their proofs. The goal is to give you a gentle introduction into these crucial areas of mathematics. Several worked-out examples will help us to get you feel comfortable when encountering mathematical statements in the future.

Before we start, let us talk about mathematical statements and their proofs from a general viewpoint. From the previous chapters, you should already know that in math, whenever you want to rigorously talk about something, you have to first formally define it. Recall that the goal is to be as precise as possible, leaving any ambiguity or doubt behind. For this purpose, a definition is used. You have already seen several examples of a definition.

After you have defined what you need, you may start reasoning about the introduced notions. For example, after you define set union, intersection, and complement (see Section 2.1), you may start wondering if there is a relation between these three operations. A relation that is not explicitly defined, but rather emerges from the definitions. And, if you do some thinking, or search the books or wikipedia, you will see that they are connected by so-called *De Morgan's laws* [2]. One of these laws says that complementing a union of two sets is the same as complementing both of them separately and then taking their intersection. In symbols, for two sets P and Q, it holds that

$$\overline{P \cup Q} = \overline{P} \cap \overline{Q}$$

Note that the definition of union, intersection, and complement does not mention any such law explicitly. However, the definition implies some facts, and these imply other facts, and so on until we arrive at De Morgan's laws.

This brings us to the first new notion that you will learn in this chapter—a *theorem.* It is a written mathematical statement that is true and has been proved. To prove something means writing its *proof.* A proof is is a convincing demonstration that some mathematical statement is necessarily true. De Morgan's laws can be written as a theorem and formally proved from the definition of union, intersection and complement.

The present chapter is organized as follows. First, Section 4.1 begins by giving reasons why is proving important. Even though proofs can be fun to discover and/or construct, students sometimes tend to perceive proofs as a necessary evil. However, you will see that there is more behind proofs that it may seem. Then, Section 4.2 talks about the general layout of mathematical statements and their types. You will learn how to write theorems and what are the differences between a theorem, lemma, and corollary. After that, in Section 4.3, you will see that there are several types of proofs and learn how to write each of them. Finally, Section 4.4 presents several examples of mathematical statements and their proofs from the theory of formal languages. Since this text is focused on the rudiments of formal language theory, mathematical principles are best to be shown applied in this particular area.

4.1 Why is Proving so Important?

You may wonder: "Why do we need proofs? What makes proving something we cannot live without?" In this section, we try to answer these questions by giving several reasons for proving.

I. Proofs assure us that what we do is right. Everyone makes mistakes. We are just humans. Proving means checking that our reasoning is correct. In math, we have got a huge advantage over, say, other sciences like physics. The advantage is that in mathematics, we make the rules, so we can prove that what we do is absolutely true. On the other hand, in physics, after doing a sufficient number of experiments, you may say that when you drop an apple, it will always fall down. However, you can never be certain, even though this may sound unimaginable[1]. In mathematics, however, a correct proof is something you can count on. Given this opportunity, we should embrace it and prove our results.
II. Proofs convince people. Some mathematical statements are obvious so the majority of people believe them without thinking twice. Probably no one will disagree that $a(b+c)$ is the same as $ab+ac$. On the other hand, there are statements which are not so apparent. As an example, consider the statement that there are infinitely many prime numbers. Without proving such a statement, you may be in doubt whether, by a chance, there is a prime that is greater than all the other primes. Why not, right? We are going to prove this statement later in this chapter so you can believe it and sleep well at night.
   This brings us to another way how to put it: proving is believing. When someone tells you not to drink milk, you would like some arguments why you should not do

[1] One of the most used argument against definitive knowledge is called a *black swan argument*, introduced by one of the greatest philosophers of science and theory of knowledge, Karl Popper: No matter how many white swans you see, you cannot be certain that all swans are white. And indeed, before the discovery of Australia, people in the old world believed that all swans are white. This statement, confirmed by a vast number of empirical observations, was invalidated by an observation of black swans living in Australia and New Zealand.

that. The same situation is with mathematical statements. Indeed, one needs a proof, which is an argument why that statement holds true.

III. Proofs save time and money. Lets say that your employer wants you to develop a method of solving a certain problem. For example, he or she may want you to write a program which can separate code from data in binary executable files (this would be really useful in disassembling and reverse compilation). If you are knowledgeable, you may immediately see that putting effort into writing such a program is a waste of time. Indeed, such a program could then be used to solve the so-called *halting problem*, a problem that has been proved to be unsolvable [5].
As another example, imagine you are writing database software for storing indexing a huge amount of data (for example, to index the Internet). Before your company spend billions of dollars for hardware, you want to be sure that your software solution (not yet tested on this huge amount of data) will scale well. A mathematical proof (in the area of time and space complexity) is the best way to ensure the validity of your solution and choice of hardware before spending a large amount of (potentially wasted) money.

IV. Proving is learning. When you were in high school, you typically learned by examples. Sometimes, however, examples may be insufficient or even misleading. When you want to fully understand how something works, reading proofs and writing them is the best way. Theorems just state *what* holds true. Proofs, on the other hand, show *why* is something true.
Furthermore, by proving, you learn how to reason carefully. This comes handy in many real-life areas. For example, when debating, you may find out that the arguments your opponent use are flawed. Careful reasoning becomes handy also in your programming practice because you want your programs to be correct. Testing may assist you, but its usability is limited (as you may know, by performing tests, you cannot prove that a program is correct [8]). Careful programming is the very first step towards correct programs.

V. Last, but certainly not least, proofs are fun :-). This particular reason may sound weird to you, but writing a proof is like solving a logical puzzle. Once you know the rules, you can focus on combining the different possibilities and discovering the solution. The situation in mathematics with proofs is analogous—once you know the basics, you may daringly start to tackle various mathematical problems. What a lovely way to spend an evening :-).

4.2 Layout and Types of Mathematical Statements

In mathematics, there exist several types of statements. You probably have already heard of things like theorems, lemmas, and axioms. After reading this section, you will understand what all of these mathematical statements mean, how to read them, and how to use them.

4.2.1 General Layout of a Statement

A general layout of a mathematical statement, including its proof, is the following:

Statement 4.1. *Formal wording of the statement.*

Proof. Argumentation that the statement is true. □

As you can see, we begin by saying that we are going to state a *statement*. This can be either a theorem, lemma, or any other type of a mathematical statement, discussed later in this section. We usually number statements for better reference. Indeed, when referring to a statement, it suffices to write "Statement 4.1".

Then, we write down the statement. For example, the wording can be "*Every integer can be written as a sum of two integers*". Traditionally, statements are written in italics so we can distinguish what is a part of the theorem and what is not.

If the statement is not adopted from other sources, like a book or a journal paper, we should include its proof. A proof usually starts with the word *Proof*, followed by a reasoning why it is true, and ends with the so-called *Q.E.D symbol* "□". Q.E.D is an acronym of the Latin phrase *quod erat demonstrandum*, meaning *which had to be demonstrated*. A yet other use of this symbol is that it clearly says where the proof ends so the text that follows it cannot be falsely mistaken as a part of the proof.

4.2.2 The Basics: Theorem, Lemma, and Corollary

We begin our tour around the types of mathematical statements by a note. Even though we will talk about theorems, lemmas, and corollaries, the distinction between them is purely subjective, and when comes to importance, all of them have the same weight. Therefore, as a rule of thumb, if you find yourself unsure what type of a statement you should use when writing mathematical text, use simply a theorem. That being said, let the tour begin.

A *theorem* is the most basic type of a statement that is proved using rigorous mathematical reasoning. Usually, theorems are regarded as the most important results. An example of some well-known theorems is the Pythagorean theorem [6], formally stated next.

Theorem 4.1. *For a right triangle with legs a and b and hypotenuse c, $a^2 + b^2 = c^2$.*

As another example, consider the following theorem from automata theory.

Theorem 4.2. *For every finite automaton, there is an equivalent regular expression and vice versa.*

A *lemma* is a minor result whose purpose is to help in proving a theorem. For example, to prove Theorem 4.2 above, we may first prove the following two lemmas, and then say that the theorem follows from them.

Lemma 4.1. *For every finite automaton, there is an equivalent regular expression.*

Lemma 4.2. *For every regular expression, there is an equivalent finite automaton.*

Lemmas are usually used for decomposing a complex proof to smaller subparts. If you come from a programming background, you can think of lemmas as some sort of mathematical equivalent to functions and procedures in programming languages. Hence, a lemma is usually used as a stepping stone to a theorem.

However, on the other hand, taking the note from the beginning of this section in account, it should not surprise you that some of the most powerful statements in mathematics are known as lemmas, including Zorn's Lemma [11], Bézout's Lemma [1], Gauss' Lemma [4], and many others. They are known as lemmas because they are usually used as mathematical tools for proving other results.

Sometimes, the proof of a theorem can have some unexpected consequences. Thus, by proving the theorem, we also obtain some other, originally unintended, results. A result of this kind is called a *corollary*. We then often say that "from Theorem X, we obtain the following corollary". As an example, consider Theorem 4.2. From this theorem, we immediately obtain the following corollary.

Corollary 4.1. *Finite automata and regular expressions define the same family of languages.*

Corollaries may also have proofs. In such cases, these proofs are usually very short.

4.2.3 Where Everything Starts: Axiom

When learning about mathematical statements, students usually wonder about the following question. To prove a theorem, we have to use previously established results, such as other theorems. However, if this is true, how the very first theorem has been proved? The answer is that these "first theorems" were not proved. Instead, we find it reasonable enough to assume them to be true in the particular environment (domain) we are analysing. We do this assumption in order to simplify the environment as much as possible.

Example 4.1. For example, in order to estimate the travelling time of a car from one city to another, it is enough for us to assume the laws of Newtonian physics. However, in order to get any meaningful estimate for the travelling time of a space rocket from Earth to Mars, we have to assume the more complex theory of relativity.

In mathematics, the theorems assumed to be true are called *axioms*, and they are the starting point of reasoning. All the other theorems are just consequences of the axioms. One can say that mathematics is a big game of what-ifs—we ask what would be the consequences (that is, what theorems would be true) if the given axioms were true. In mathematical text, axioms are usually given in the form of a definition.

As an example, consider Euclidean geometry, which you have already encountered in high school. It is based on the following five axioms [3]:

1. A straight line may be drawn from any given point to any other.
2. A straight line may be extended to any finite length.
3. A circle may be described with any given point as its center and any distance as its radius.
4. All right angles are congruent.
5. (*The parallel postulate.*) If a straight line intersects two other straight lines, and so makes the two interior angles on one side of it together less than two right angles, then the other straight lines will meet at a point if extended far enough on the side on which the angles are less than two right angles.

As the last axiom can be somewhat tedious to decode (remember, that reading mathematics is like reading a source code of a program—one have to think about each word and each line), an intuitive explanation is presented in Figure 4.1.

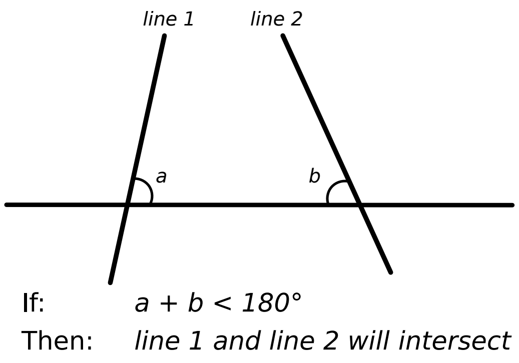


Fig. 4.1. *The parallel postulate*: If the sum of the angles is less than 180°, the lines will intersect.

These axioms formally describe our intuition about the geometry on a flat plane, such as paper. One of the consequences of these axioms is that the sum of the angles of a triangle is always 180°.

On the other hand, if we drop some of the axioms and add some other axioms, we can end up with a non-Euclidean geometry, where the sum of the angles of a triangle is not always 180°. For example, consider a geometry on the surface of a sphere, as seen in Figure 4.2.

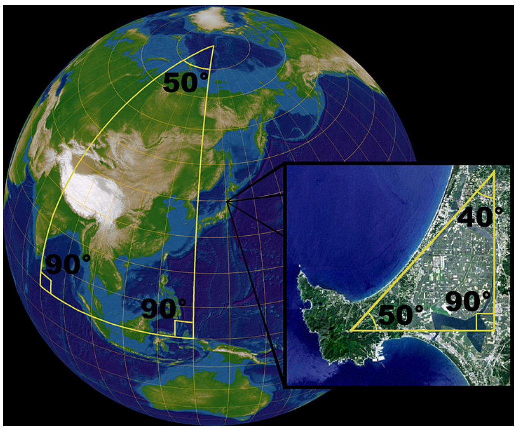


Fig. 4.2. An example of *spherical geometry* [9], where only some of the Euclidean postulates hold.The sum of the angles of a triangle in such a geometry is not always 180°. These kind of models are used, for example, in airplane navigation.

4.2.4 Summary

In this section, we have learned that there are several types of mathematical statements. Table 4.1 summarizes these types, including short notes on their usage. The table may be used as a quick reference.

Type of a statement	**Usage**
Theorem	You want to write a statement that you prove on the basis of previously established results. As a rule of thumb, if you do not know what type of a statement you should use, use a theorem.
Lemma	You want to divide a proof of a theorem into several parts, where each part is a lemma. It is usually used as a stepping stone to a theorem. There is no formal distinction between a lemma and a theorem.
Corollary	You want to write a statement that follows readily from a previous statement. There is no formal distinction between a theorem, lemma, and corollary. Use of a corollary is plainly subjective.
Axiom	A mathematical statement that serves as a starting point from which other statements are derived. You do not usually write these—they are given.

Table 4.1. Summary of mathematical statements and their usage.

4.3 Types of Mathematical Proofs

Even though each proof in mathematics is, in a sense, unique, they share some common patterns. The key pattern is the form of the proof and the way it achieves its goal—show that a given mathematical statement is correct. Based on this form, we may distinguish three basic proof techniques (also called *types* of proofs): a direct proof, a proof by contradiction, and a proof by induction. All of them are described in detail in the subsequent sections. More specifically, Section 4.3.1 discusses direct proofs, Section 4.3.2 describes proofs by contradiction and, finally, Section 4.3.3 presents proofs by induction. Although these are the most basic and maybe one of the most common used proof techniques, there are a lot more types of proofs.

If you come from a programming background, you can think of these techniques as some sort of mathematical equivalents to a *design patterns*. They serve like some kind of a template, so you do not have to adhere to them strictly. Furthermore, you can mix them in various ways or use one technique inside the other.

Throughout this section, all types of proofs are first demonstrated on rather simple mathematical theorems. Proofs from formal language theory are then given in Section 4.4. The reason is that we believe the reader should first understand the basic principles underlying these types of proofs, and then see their application in the studied area. In our case, the area is is the theory of formal languages.

4.3.1 Direct Proof

A direct proof is the most basic type of a proof. In a direct proof, we show that a statement is true by combining known facts. For example, consider a situation where we have to prove that Carl will be fired from a fictional company. Furthermore, we know the following two facts:

(1) If someone does not do their job, they will be fired.
(2) Carl does not do his job.

Notice, that these "facts" are in fact just a postulated axioms—the idealistic simplified version of the real life, where it is not always that easy to get fired. However, as we have our axioms as the basis for logical reasoning, we can now make some logical consequences from these axioms:

- By (2), we know that Carl does not do his job.
- From (1), we know that such a person will be fired.
- Hence, we have that Carl will be fired.

Next, we give a less trivial demonstration of a direct proof. Recall that an integer is *even* if it is divisible by 2 without a remainder. For example, the numbers 2, 4, 100 are all even. The following example can be really simple for some students, but you will see that giving a formal proof of an obvious statements is not always easy.

Theorem 4.3. *The sum of two even integers is itself an even integer.*

Before proving the theorem, let us first think about the proof. We have to start with what we have. What do we have, exactly? Well, we have two even integers, a and b. Now, what are we supposed to prove? We have to prove that $a + b$ is an even integer. How should we prove it? We need to evaluate the properties of even numbers, and take them into consideration. One such a property is that since a and b are both even, we can divide them by 2 without a remainder. Thus, we can rewrite them in a form such that when we add them, we obtain an even number. All right, let us prove it.

Proof. Let a and b be two even integers. Since they are even, they can be written in the forms $a = 2x$ and $b = 2y$ for some integers x and y, respectively. Then, $a + b$ can be written in the form $2x + 2y$, resulting in the following equation:

$$a + b = 2x + 2y = 2(x + y)$$

From this, we see that $a + b$ is divisible by 2. Hence, $a + b$ is an even integer, and the theorem holds. □

The most difficult part of a proof to figure out is the reasoning. Sometimes, you have a large number of possibilities you have to explore, and most of these possibilities lead nowhere. Other times, you just happen to have an insight and see the right path from the beginning. Unfortunately, there is no textbook that would enable you to prove all theorems. Therefore, in a sense, mathematics is art—an art of proving theorems.

4.3.2 Proof by Contradiction

A *proof by contradiction*, also known by its Latin name *reductio ad impossibilem*, can be somewhat unintuitive at the first look, but after a brief study, you will see that it is simple and elegant. This proof technique is based on the following two basic rules of mathematical logic:

1. Any mathematical statement is either true or false.
2. If a statement is true, its negation is false.

A proof by contradiction works as follows:

> To prove that a statement A holds, we start by assuming that A does not hold.

This is the unintuitive part of the proof—it starts by our insight into the problem. From this insight, we can guess that the statement A is true. However, unfortunately, we are not able prove it directly. Thus, we assume that A does not hold, but only because we will explore the consequences of that assumption, hoping that we will arrive to some contradiction. The basic idea behind this is that by negating the original A, it is now much easier for us to create a logical argumentation. Therefore, we combine some known facts, just like in a direct proof, which finally results into a contradiction, like $1 = 2$, or a contradiction on the initial assumption.

However, this contradiction proves our assumption of "A does not hold" to be completely false! Therefore, by the second rule of mathematical logic, the statement "A does hold" have to be true.

We proceed by giving an example. Consider the following proof of the statement that there is an infinite number of primes. Recall that a *prime* number is a natural number greater than 1 that has no positive divisors other than 1 and itself. Notice that by definition, 2 is a prime, which is also the only even prime number. For example, the numbers 2, 3, 5, 17 are all primes.

Theorem 4.4. *There are infinitely many primes.*

Note that as we want to prove that there are infinitely many primes, we cannot list them all to prove the theorem. Also, although prime numbers have a lot of properties, these properties are complex and not so easy to work with. However, if we negate the theorem, we will end up with the statement "There is a finite number of primes". It is much easier for us to work with this statement—we may try to find the largest prime number. Of course, as we believe that there really is an infinite number of primes, there is no largest prime number, so we hope to obtain some contradiction.

Proof. To obtain a contradiction, we will assume that there exist only finitely many prime numbers

$$p_1 < p_2 < \cdots < p_n$$

Let $q = p_1 p_2 \cdots p_n + 1$ be the product of $p_1, p_2, \ldots, p_n$ plus one. Like any other natural number, q is divisible by at least one prime number (it is possible that q itself is a prime). However, none of the primes $p_1, p_2, \ldots, p_n$ divides q without a remainder because dividing q by any of them leaves a remainder 1. Therefore, there has to exist a yet other prime number than $p_1, p_2, \ldots, p_n$, which is a contradiction with the initial assumption. Therefore, there are infinitely many primes. □

As yet another example, consider the following proof of the statement that the square root of 2 is not rational. Recall that a number is *rational* if it can be written in the form $\frac{a}{b}$, where a and b are two integers such that a does not divide b and vice versa.

The first thing we need to do is to realize that irrational numbers are more complex then rational numbers. Therefore, it would be more convenient for us to negate the statement, claiming that $\sqrt{2}$ is rational, and then work with the properties of rational numbers to find some contradiction. So, lets look at some interesting properties:

1. Each rational number can be expressed as $\frac{a}{b}$, where a and b are the smallest possible numbers. For example, the fractions $\frac{4}{8}$, $\frac{3}{6}$, and $\frac{2}{4}$ can be all written as $\frac{1}{2}$.
2. Thus, a and b cannot be both even because then we could divide them by 2 and get smaller numbers.
3. If $x = y$, then $x^2 = y^2$.
4. If x is even, then x^2 is also even; if x is odd, then x^2 is also odd.

Now, we can combine these properties to obtain a contradiction. So, let us write it down in a formal manner.

Theorem 4.5. $\sqrt{2}$ *is not rational.*

Proof. To obtain a contradiction, assume that $\sqrt{2}$ is a rational number. Since it is rational, it can be expressed as $\frac{a}{b}$, where a and b are the smallest possible integers (recall the first property). Then, at least one of them has to be odd (recall the second property). However, if $\frac{a}{b} = \sqrt{2}$, then $(\frac{a}{b})^2 = (\sqrt{2})^2$ (recall the third property)—that is $\frac{a^2}{b^2} = 2$, so $a^2 = 2b^2$. Therefore, a^2 has to be even. Since the square of an odd number is odd (recall the fourth property), we have that a is even. This means that b has to be odd (recall the second property: one of them has to be odd).

So, we have that a is even—that is, $a = 2x$ for some integer x. This implies that a^2 is a multiple of 4—that is, $a^2 = 4x^2$. Together with the equality $a^2 = 2b^2$, we have that $2b^2 = 4x^2$, which means that $2b^2$ is a multiple of 4. Therefore, $b^2 = 2x^2$, so b^2 is even and, therefore, b has to be also even (recall the fourth property).

Hence, b is both odd and even at the same—a contradiction. Therefore, the initial assumption that $\sqrt{b}$ is rational, has to be false. □

4.3.3 Proof by Induction

A *proof by induction* is typically used to prove that a statement holds for all natural numbers. As there are infinitely many natural numbers, we cannot prove some statement A for all of them just by taking one after another. Every proof has to have a finite number of steps! However, fortunately, natural numbers have one very strong property—each number has a successor (by a successor, we mean the next number: 1 is the successor of 0, 2 is the successor of 1, 3 is the successor of 2, etc.). And, beginning at zero, by following the line of successors, we can get to any number.

We can use this property for our proof—first, we prove the statement A for 0, and then we show that the truth value of A is spreading through its successors. That is, if A holds for 0, it has to hold also for 1. However, now, as A holds for 1, it has to hold also for the successor of 1—that is 2. So it holds for 3, and then for 4 and so on.

To prove that A holds for all natural numbers, it is sufficient to prove two things: to show the *spreading nature* of the statement A, and to show the starting point of the spreading (usually the number 0). Formally, we need to prove the following mathematical statements:

1. A holds for 0 (the starting point, called *basis*).
2. If A holds for n, then it also holds for $n + 1$ (the spreading nature, called *induction step*).

It is important to note that the induction step does not tell us whether A holds for n or not. It just tells us what would happen if A would be true for n. It is truly proving just the capability of A to spread, and it needs the basis to actually spread through the natural numbers. Indeed, since the statement holds for 0, then by using the second step, we know it also holds for 1. Using the second step again, we see that it holds for 2, and so on, ad infinitum.

Before delving into more details, let us give a simple example of mathematical induction in practice. For every natural number n, let $S(n)$ denote the sum of all the numbers 0, 1, 2, …, n. In symbols,

$$S(n) = \sum_{0 \le i \le n} i$$

Theorem 4.6. $S(n) = \frac{n(n+1)}{2}$

Proof. We prove this theorem by induction.

Basis: We show that the statement holds for 0. This means we have to prove that

$$0 = \frac{0(0+1)}{2}$$

Since the right-hand side can be simplified to 0, we have that $0 = 0$, so the basis holds.

Induction Step: In the induction step, we have to show that if the statement holds for $S(n)$, then it holds for $S(n+1)$. To this end, assume that it holds for $S(n)$—that is, we ask the question "What would happen if it holds for n?" in a mathematical way. Then, to prove that it holds for $S(n+1)$, we have to prove that

$$(0+1+2+\cdots+n)+(n+1) = \frac{(n+1)((n+1)+1)}{2}$$

By using the assumption that $S(n)$ is true, the left-hand side of the equation can be rewritten to

$$\frac{n(n+1)}{2} + (n+1)$$

This expression then can be rewritten in the following way:

$$\begin{aligned}\frac{n(n+1)}{2} + (n+1) &= \frac{n(n+1)+2(n+1)}{2} \\ &= \frac{n^2+n+2n+2}{2} \\ &= \frac{(n+1)(n+2)}{2} \\ &= \frac{(n+1)((n+1)+1)}{2}\end{aligned}$$

This implies that $S(n+1)$ holds. Since we have proved both the basis and the induction step, by the principle of induction, the theorem holds. □

As you can see from the previous example, to prove the induction step, we used the assumption that the statement holds for $S(n)$. This is typical for induction proofs. This assumption is used purely in the "what-if" manner—if the theorem holds for n, would it hold also for $n+1$? Notice that if we did not use this assumption, it would have been difficult to complete the induction step. Try it by yourself to see the difficulty.

Usually, the induction basis is proved for 0. However, the induction proof works even if we start from a different number. For example, in some situations, the statement does not hold for some small numbers, like 0, 1, and 2. In such a case, we may start by proving the basis for 3, and then continue as we are used to.

When advantageous, we may use a proof by induction inside of a direct proof or a proof by contradiction. Some people do not even recognize proof by induction as a separate type of a proof—they put it into the category of direct proofs.

You can also note that the induction proof really needs just the successor property. Thus, it can be used on any mathematical structure satisfying this property, not just natural numbers. Examples of mathematical structures with successor property include trees or graphs, which are widely used in computer science.

4.4 Examples from Formal Language Theory

In this section, we give many examples of theorems in formal language theory, including their detailed proofs. The goal of the section is twofold. First, you will see how mathematical statements are used to state things formally. Second, you can follow the train of thoughts leading from the statement's wording to its complete proof.

Since the reader of this document is assumed to either be a student in some course dealing with formal language theory or to be interested in this area, mathematical principles are best to be shown applied in this particular area. If you are not familiar with some of the notions we use throughout this section, we kindly refer you to [18].

4.4.1 Concatenation of Languages

First, we consider languages and operations over them. Recall that a language is a set of strings. Since every language is a set, all common operations over sets apply also to languages. There are also some special operations that apply only to languages. For example, consider concatenation. Recall that when concatenating two strings x and y, we obtain a new string xy. Furthermore, when we concatenate two languages, L_1 and L_2, we obtain a new language, where we concatenate each string of L_1 with each string of L_2. In symbols,

$$L_1L_2 = \{xy : x \in L_1, y \in L_2\}$$

The above symbolic definition of a concatenation of two languages reads as follows. "L_1L_2" denotes the concatenation of L_1 and L_2. "$L_1L_2 =$" means that we are going to write what the concatenation equals to. Finally, the right-hand side of the equation says that this concatenation is composed of strings of the form xy, where x is a string from L_1 and y is a string from L_2. Furthermore, the definition says that we need to include all strings that satisfy the property of $x \in L_2$ and $y \in L_2$.

Also, we have to recall the notion of the empty string, which is denoted by ε. This is a special string with no symbols. What makes it special? There is no other string with this property. Indeed, if you take any other string, it has to contain at least one symbol.

Our task will be to prove that when concatenating the language containing just the empty string to any other language L, we obtain precisely L. Furthermore, we will prove that it does not matter whether we concatenate the language containing just the empty string to L from the left or from the right.

Since the above description is rather informal, and mathematical statements are a formal business, we first have to try to write down what we are about to prove mathematically. Lets do it. The empty string is denoted by ε. Therefore, the language containing just the empty string is denoted by $\{\varepsilon\}$. A concatenation of this language to a language L is denoted by $L\{\varepsilon\}$. The task assignment says that this concatenation should be equal to L. This can be written as $L\{\varepsilon\} = L$. Finally, since it should not matter if we concatenate L to $\{\varepsilon\}$ or vice versa, we can write $L\{\varepsilon\} = \{\varepsilon\}L = L$. This gives use the following wording of the theorem we will prove.

Theorem 4.7. *For every language L, $L\{\varepsilon\} = \{\varepsilon\}L = L$.*

Before delving into a formal proof, let us stop for a second and think about it. How to prove this theorem? Well, the easiest way would be to divide it into two steps. First, we prove that $L\{\varepsilon\} = L$, and then we prove that $\{\varepsilon\}L = L$. Then, since $L\{\varepsilon\} = L$ and $\{\varepsilon\}L = L$, $L\{\varepsilon\} = L = \{\varepsilon\}L$, which is the same as $L\{\varepsilon\} = \{\varepsilon\}L = L$. Nice. Now we have a way of proving the statement.

Since we are going to split the proof into two parts, this is a typical situation which calls for using a stepping stone. Can you guess what the stepping stone will be? Yes, we will use a lemma. From Section 4.2.2, we know that a lemma is a proven statement which is used to prove a larger result rather than as a statement of interest by itself. We can divide the theorem into two lemmas, prove each of them separately, and then put them together to prove the theorem. In this way, we will be doing a single step at a time, thus we can always focus on a single thing.

Let us start with the first lemma.

Lemma 4.3. *For every language L, $L\{\varepsilon\} = L$.*

Proof. Where should we start? Well, it is best to start from the beginning, which means that we start with a language L. So, let L be a language. Now, we have to prove that $L\{\varepsilon\} = L$. The right-hand side cannot be simplified, so we should take a look at the left-hand side. To step forward, we can try to rewrite the left-hand side according to the definition of a concatenation, which is

$$L\{\varepsilon\} = \big\{xy : x \in L, y \in \{\varepsilon\}\big\}$$

Since there is only a single string in $\{\varepsilon\}$, we can rewrite the new right-hand side in the following way:

$$\big\{xy : x \in L, y \in \{\varepsilon\}\big\} = \big\{x\varepsilon : x \in L\big\}$$

By recalling that $x\varepsilon = x$, we can simplify it as follows:

$$\big\{x\varepsilon : x \in L\big\} = \big\{x : x \in L\big\}$$

Now, the right-hand side is only a fancy way to write L, so

$$\big\{x : x \in L\big\} = L$$

Since we have been rewriting $L\{\varepsilon\}$, we have that $L\{\varepsilon\} = L$. Hence, we are done, and the lemma holds. □

Let us move to the second lemma.

Lemma 4.4. *For every language L, $\{\varepsilon\}L = L$.*

Proof. We can prove this lemma in an analogical way we have proved the first lemma. Indeed, the only difference is that when simplifying the left-hand side, we put $x = \varepsilon$ instead of $y = \varepsilon$. To improve your proving skills, try to formulate the proof by yourself. $\square$

Now, we can put these two lemmas together to prove the theorem.

Theorem 4.8. *For every language L, $L\{\varepsilon\} = \{\varepsilon\}L = L$.*

Proof. This theorem follows directly from Lemmas 4.3 and 4.4. Indeed, from Lemma 4.3, we have that $L\{\varepsilon\} = L$, and Lemma 4.4 implies that $\{\varepsilon\}L = L$. Hence, $L\{\varepsilon\} = L = \{\varepsilon\}L$, which is the same as $L\{\varepsilon\} = \{\varepsilon\}L = L$. $\square$

With regard to the terminology of Section 4.3, all the proofs above are examples of a direct proof. Next, we turn our attention to a proof by contradiction.

4.4.2 Pumping Lemma

As you may know, a pumping lemma for regular languages states that if a language is regular, then all sufficiently long strings can be "pumped" so that all of these "pumped" strings belong to the language.

The basic idea behind this lemma is derived from the fact that there is a finite automaton for each regular language. As a finite automaton has a finite number of states, it has to loop for any accepted string that is long enough (in fact, longer than the number of states). However, as there is a loop in the automaton, it can loop for any number of times and accept even longer strings. This looping during accepting is equivalent with "pumping" the corresponding part of the string.

Formally, the lemma can be stated in the following way:

Lemma 4.5. *Let L be a regular language. Then, there exists a constant $k \geq 1$ such that if $z \in L$ and $|z| \geq k$, then there exist u, v, w such that $z = uvw$, and the following three conditions are satisfied:*

(1) $v \neq \varepsilon$,
(2) $|uv| \leq k$, and
(3) for each $m \geq 0$, $uv^m w \in L$.

Let us try to read the wording of the lemma. It says that if we take any regular language L, then there is definitely a positive constant k such that if we take any string in the language that is at least k symbols long, then we can decompose it into three substrings, u, v, and w (the most important part for us is v because it can be pumped). Furthermore, the conditions (1) through (3) are satisfied. (1) says that the middle substring, v, is non-empty. (2) says that the length of uv is at most k. Finally, (3) is the pumping condition which tells us that if we take string where v is powered to any non-negative integer, then such a string is also in the language.

The first question that students usually ask is: "What is the exact value of k? 10? 250?" The answer is that it depends on the language (as it is usually the number of states in its respective automaton[2]). Moreover, as we will see, the exact value of k is not important. What is important is that if a language is regular, there is always such a constant.

Notice that if the language is not regular, then this lemma says nothing about it. Indeed, the decomposition into uvw and the three conditions hold only if the language is regular. Moreover, there are some non-regular languages which satisfy the lemma. Therefore, by using this lemma, we can never prove that a language is regular. If we want to prove that a language is regular, we construct a finite automaton that accepts it, or a regular expression that defines it.

The pumping lemma is typically used to prove that a language is *not* regular by using a proof by contradiction. This is the reason it is called *lemma* and not *theorem*—it is primarily used as a tool in proofs. For example, take some language L which we believe is not regular. To prove that, we will proceed as follows. First, to obtain a contradiction, we will assume that L is regular. Then, by the pumping lemma, there exists a constant such that for all sufficiently long strings, there is a decomposition satisfying (1) through (3). We then show that for every sufficiently long string, such a decomposition does not exit. This is, however, a contradiction with the pumping lemma, which says that it always has to exist. Therefore, we obtain a contradiction with the assumption that L is regular, so, by the principle of a proof by contradiction, L is not regular.

If you do not understand it right now, do not worry. We will now go through a complete example. The assignment is to prove that the language containing strings having the same number of as and bs is not regular, as stated by the following theorem.

Theorem 4.9. *The language* $L = \{a^n b^n : n \geq 0\}$ *is not regular.*

Before we go and prove the theorem, let us first think about the reason the language is not regular. What does it mean when a language is not regular? It means that we cannot construct a finite automaton that accepts it. If such an automaton did exist, how would it work? We would have to remember how many as we have read so when we start reading bs, we can check that their number matches. Where can we store the number? The only way is to use states. So, we may create states denoting we have read a single a, two as, and so on. However, as there is no limit on n, we would need an infinite number

[2] There can be more than one automaton for every language. However, it is not important which automaton we chose for k as long as the automaton accepts the language in question.

of states. This conflicts with the definition of a finite automaton. Indeed, its names stems from the fact that it has a finite number of states. Alright, let us quit talking and do some proving.

Proof. We have to show that L from the theorem is not regular. We will proceed by contradiction. To obtain a contradiction, we assume that L is regular. Then, by the pumping lemma, there exists a constant $k \geq 1$ such that if $z \in L$ and $|z| \geq k$, then there exist u, v, w such that $z = uvw$, and the following three conditions are satisfied:

(1) $v \neq \varepsilon$,
(2) $|uv| \leq k$, and
(3) for each $m \geq 0$, $uv^m w \in L$.

The first thing we have to handle is that we do not know the exact value of k. So, one may ask: "How can we choose a string from L that is longer than k when we do not know the value of k?" Quite simply—we put $z = a^k b^k$. From the definition of L, we know that z belongs to L, and since k is greater or equal to 1, the length of z is certainly greater than k. To be precise, $|z| = 2k$.

So, we have z whose length is greater than k. Now comes the tricky part. The pumping lemma states that z can be decomposed so that (1) through (3) are satisfied. So, let $z = uvw$ such that (1) through (3) hold. By (1), v is non-empty. By (3), uv^2w should be in L. However, since $z = a^k b^k$, no matter what v is, if it appears twice, as in uv^2w, the resulting string cannot belong to L. Indeed, there are the following three possibilities (depicted in Figure 4.3) what v is:

(i) v contains just as. Then, in uv^2w, there will be more as than bs, which violates the definition of L.
(ii) v contains just bs. As in the previous case, in uv^2w, there will be more bs than as, so the string cannot be in L.
(iii) v begins with some as and ends with some bs. Then, however, in uv^2w, we as are intermixed with bs, which is also a violation of the definition of L.

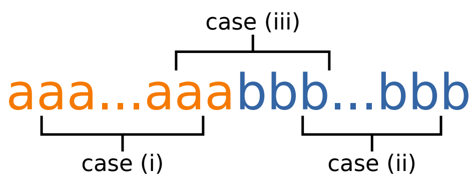


Fig. 4.3. Three possible cases of dividing the string $a^n b^n$ into three parts u, v, w. Only v is depicted in the picture for each case as u is everything that precedes v, and w is everything that follows after v.

Based on the previous argumentation, we see that no such decomposition is possible. However, the pumping lemma tells us that it has to exist. This is the contradiction we have been looking for. Therefore, the assumption that L is regular does not hold, and so we have just proved that L is not a regular language. $\square$

Let us now go back to Section 4.2. In there, we have learned that a corollary is a mathematical statement that readily follows from a previous statement. To show this applied in practice, consider the theorem we have just proved. It states that the language $L = \{a^n b^n : n \geq 0\}$ is not regular. What does it mean? We know that every regular language can be accepted by a finite automaton. So, from the theorem, we see that there cannot exist any finite automaton that accepts L. This can be stated as a corollary, as it follows directly from the proof of the previous theorem and need no additional proof or argumentation.

Corollary 4.2. *There is no finite automaton that accepts $L = \{a^n b^n : n \geq 0\}$.*

Since now, we have successfully used a lemma, a theorem, and now a corollary. In terms of proofs, we have used a direct proof and a proof by contradiction. What leaves is a proof by induction. This brings us to the next subsection.

4.4.3 Operations Over Strings

Consider strings. What is a string? A sequence of symbols. Therefore, every string can be written in the form $a_1 a_2 \cdots a_n$, where $n \geq 1$, and every a_i is a symbol. The *reversal* of such a string is obtained by writing the symbols in the reverse order. That is, the reversal of $a_1 a_2 \cdots a_n$ is $a_n a_{n-1} \cdots a_1$. For example, the reversal of the string *abbcd* is *dcbba*. The reversal of a string x will be denoted by $\mathrm{rev}(x)$.

When proving theorems, it is useful to know previously established theorems, which you can use to simplify your proofs. You may find such theorems in books or journal articles. We will use a theorem by ourselves in the following proof. The theorem we are going to use states that when reversing a string, we obtain the same result as by splitting it into two parts, reversing both of them, and concatenating them in a reverse order.

Theorem 4.10. *Let u and w be two strings. Then, $rev(uw) = rev(w)rev(u)$.*

A yet other operation over strings is power. The *power* of a string x is defined as follows. For 0, we define $x^0 = \varepsilon$. That is, x to the power of 0 equals the empty string. Then, for $n \geq 1$, we just put $x^n = xx^{n-1}$. This is an example of a so-called recursive definition. For example, $x^1 = x$, $x^2 = xx^{2-1} = xx^1 = xx$, and

$$x^3 = xx^{3-1} = xx^2 = xxx^{2-1} = xxx^1 = xxx$$

Another theorem which we will need is the following one:

Theorem 4.11. *Let x be a string and n be a natural number. Then, $x^n = x^{n-1}x$.*

Now that we are familiar with all the terminology, we can proceed to an example. In this example, our task is to prove that given a string x, for every natural number n, it does not matter if we first exponentiate x to n and then reverse it or vice versa. If we rewrite this task by using symbols, we obtain the following equation:

$$\operatorname{rev}(x^n) = \operatorname{rev}(x)^n$$

Therefore, our task is to prove the following theorem.

Theorem 4.12. *Let x be a string. Then, for every natural number n,* $\operatorname{rev}(x^n) = \operatorname{rev}(x)^n$*.*

Proof. Alright, so we have to start with a string. Let x be a string. Now, notice that we cannot simply just choose some number and put it as the value of n because we are required to prove it for every natural number. By recalling the contents of Section 4.3, for theorems ranging over natural numbers, a proof by induction is the choice we are going to make.

Our proof will be done by induction on n, as this is the variable the theorem is ranging over. We start with a basis. The first natural number is zero, which means we have to prove that $\operatorname{rev}(x^0) = \operatorname{rev}(x)^0$. Since $x^0 = \varepsilon$ by the definition of power, the left-hand side can be simplified to $\operatorname{rev}(\varepsilon)$, which equals ε. Then, by using the same reasoning with the definition of the zero power, the right-hand side of the equation can be simplified to ε. Indeed, as any string to the power of 0 is the empty string, it does not matter what x actually is; the result will always be the empty string. So, we have that $\varepsilon = \varepsilon$, and the basis holds.

Now the induction step. Recall that in the induction step, we have to prove that if the statement holds for n, then it holds for $n+1$. To this end, assume that the statement holds for n—that is, $\operatorname{rev}(x^n) = \operatorname{rev}(x)^n$. Then, to prove that it holds for $n+1$, we have to prove the following equation:

$$\operatorname{rev}(x^{n+1}) = \operatorname{rev}(x)^{n+1}$$

In what follows, we have to somehow utilize the assumption that the statement holds for n—that is, we assume that $\operatorname{rev}(x^n) = \operatorname{rev}(x)^n$, and we are proving the statement for $n+1$. By Theorem 4.11, $\operatorname{rev}(x^{n+1})$ can be rewritten to $\operatorname{rev}(x^n x)$. Furthermore, by the definition of the power operator, the right-hand side $\operatorname{rev}(x)^{n+1}$ can be rewritten to $\operatorname{rev}(x)\operatorname{rev}(x)^n$. Hence, we now have that

$$\operatorname{rev}(x^n x) = \operatorname{rev}(x)\operatorname{rev}(x)^n$$

By using the induction assumption that $\operatorname{rev}(x^n) = \operatorname{rev}(x)^n$, we may rewrite $\operatorname{rev}(x)\operatorname{rev}(x)^n$ to $\operatorname{rev}(x)\operatorname{rev}(x^n)$, so we end up with

$$\operatorname{rev}(x^n x) = \operatorname{rev}(x)\operatorname{rev}(x^n)$$

Now comes the right time to use the theorem we were talking about. Reconsider Theorem 4.10. By this theorem, when $u = x^n$ and $w = x$, $\operatorname{rev}(x^n x)$ can be rewritten to $\operatorname{rev}(x)\operatorname{rev}(x^n)$, which results in

$$\mathrm{rev}(x)\,\mathrm{rev}(x^n) = \mathrm{rev}(x)\,\mathrm{rev}(x^n)$$

As you can see, both sides of the above equation are the same. Hence, $\mathrm{rev}(x^{n+1}) = \mathrm{rev}(x)^{n+1}$, which completes the induction step, and the theorem holds. □

5 Conclusion

The principle goal of this document was to give you a gentle introduction into the mathematical foundations of formal language theory. The secondary goals included a demonstration of the usefulness of mathematical notation and rigorous methods, notes on how to read and write your of definitions and statements, and why are mathematical proofs of major importance. In this concluding chapter, we very briefly review all that we have talked about and give you some references for your further studies.

First, in the introductory Chapter 1, we have seen the reasons why mathematics matters, why making abstractions by means of modelling may give use an advantage, why to learn core topics before hot topics, and why you should be interested in the subject of the present document.

After that, Chapter 2 provided the very basic mathematical notions that underly the theory of formal languages. As a running example, we have selected one of the simplest (but widely used) models—a finite automaton. In a step-by-step way, we have started with defining states, input alphabets, and final states by utilizing sets. Then, as sets were not enough or not very suitable to capture the remaining components of a finite automaton, we have visited sequences and relations. By their means, we have seen how to define the set of transition rules and the finite automaton itself—as a quintuple, consisting of a finite set of states, a finite set of input symbols, a set of rules, a designated start state, and a set of final states. Finally, we have mentioned functions, which provided us with a tool to define the deterministic variant of a finite automaton in a rather straightforward way. All in all, we have seen how all the pieces fit together in the formal definition of a finite automaton.

Chapter 3 then continued the investigation of finite automata by formalizing their process of computation and, most importantly, their accepted language. Recall that we have based the definition of computation on the direct move relation, and then the accepted language has been defined so that it contains precisely the strings that lead the finite automaton to a final state. After that, to provide a simpler definition of computation, we have studied closures. And indeed, we have seen that computation can be defined straightforwardly as the reflexive and transitive closure of the direct move relation.

Finally, in Chapter 4, we have undergone a journey through the world of mathematical statements and their proofs. Along the way, we have learned why such statements are of importance for us, what types of them are there and how to read them, why proving matters, and what basic types of proofs are there. In the end of the chapter, we have seen several applications of mathematical statements and their proofs in terms of formal language theory.

We hope that we have succeeded to fulfill all the goals and that it has been a pleasant adventure for you. Nevertheless, keep in mind that we have covered just the very fundamental basics of mathematical foundations of formal language theory. If you liked this journey through the model-building aspects of mathematics, you can continue by moving forward and starting reading mathematically oriented books or books on the theory of formal languages. For a good, more detailed treatment of mathematical foundations underlying formal language theory, consult [20]. Other suggested mathematically oriented books include [12–15]. As good introductory books to formal language theory, we recommend [16–19, 21].

Bibliography

1. Bézout's identity [online]. Last update 2012-09-23. [cit. 2012-11-08]. Available on URL: `http://en.wikipedia.org/wiki/Bezouts_identity`.
2. De morgan's laws [online]. Last update 2012-10-21. [cit. 2012-11-09]. Available on URL: `http://en.wikipedia.org/wiki/De_Morgan_Laws`.
3. Five postulates of Euclidean geometry [online]. Last update 2012-08-09. [cit. 2012-10-13]. Available on URL: `http://en.wikibooks.org/wiki/Geometry/Five_Postulates_of_Euclidean_Geometry`.
4. Gauss' lemma [online]. Last update 2012-09-25. [cit. 2012-11-08]. Available on URL: `http://en.wikipedia.org/wiki/Gauss's_lemma_(number_theory)`.
5. Halting problem [online]. Last update 2012-10-10. [cit. 2012-11-07]. Available on URL: `http://en.wikipedia.org/wiki/Halting_problem`.
6. Pythagorean theorem [online]. Last update 2012-11-08. [cit. 2012-11-08]. Available on URL: `http://en.wikipedia.org/wiki/Pythagorean_theorem`.
7. Series (mathematics) [online]. Last update 2012-12-06. [cit. 2012-12-14]. Available on URL: `http://en.wikipedia.org/wiki/Series_(mathematics)`.
8. Software testing [online]. Last update 2012-11-07. [cit. 2012-11-07]. Available on URL: `http://en.wikipedia.org/wiki/Software_testing`.
9. Spherical geometry [online]. Last update 2012-12-29. [cit. 2012-12-30]. Available on URL: `http://en.wikipedia.org/wiki/Spherical_geometry`.
10. WTF, man I just wanted to learn how to program video games [online]. Last update 2011-09-03. [cit. 2012-12-30]. Available on URL: `http://www.lolroflmao.com/2011/09/03/wtf-man-i-just-wanted-to-learn-how-to-program-video-games/`.
11. Zorn's lemma [online]. Last update 2012-10-10. [cit. 2012-11-08]. Available on URL: `http://en.wikipedia.org/wiki/Zorn_lemma`.
12. T. Gowers. *Mathematics: A Very Short Introduction.* Oxford University Press, New York, 2002.
13. T. Gowers. *The Princeton Companion to Mathematics.* Princeton University Press, Princeton, 2008.
14. R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley, Boston, 2nd edition, 1994.
15. P. Halmos. *Naive Set Theory.* Springer, New York, 1998.
16. M. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Boston, 1978.
17. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Boston, 1979.
18. A. Meduna. *Automata and Languages: Theory and Applications.* Springer, London, 2000.

19. M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, 2nd edition, 2006.
20. J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, New York, 2nd edition, 2003.
21. D. Wood. *Theory of Computation: A Primer.* Addison-Wesley, Boston, 1987.

Index