Introduction to Theory of Computation

Anil Maheshwari Michiel Smid

School of Computer Science

Carleton University

Ottawa

Canada

{anil,michiel}@scs.carleton.ca

March 23, 2017

Contents

Preface

This is a free textbook for an undergraduate course on the Theory of Computation, which we have been teaching at Carleton University since 2002. Until the 2011/2012 academic year, this course was offered as a second-year course (COMP 2805) and was compulsory for all Computer Science students. Starting with the 2012/2013 academic year, the course has been downgraded to a third-year optional course (COMP 3803).

We have been developing this book since we started teaching this course. Currently, we cover most of the material from Chapters 2–5 during a 12-week term with three hours of classes per week.

The material from Chapter 6, on Complexity Theory, is taught in the third-year course COMP 3804 (Design and Analysis of Algorithms). In the early years of COMP 2805, we gave a two-lecture overview of Complexity Theory at the end of the term. Even though this overview has disappeared from the course, we decided to keep Chapter 6. This chapter has not been revised/modified for a long time.

The course as we teach it today has been influenced by the following two textbooks:

- Introduction to the Theory of Computation (second edition), by Michael Sipser, Thomson Course Technnology, Boston, 2006.

- Einführung in die Theoretische Informatik, by Klaus Wagner, Springer-Verlag, Berlin, 1994.

Besides reading this text, we recommend that you also take a look at these excellent textbooks, as well as one or more of the following ones:

- Elements of the Theory of Computation (second edition), by Harry Lewis and Christos Papadimitriou, Prentice-Hall, 1998.

- Introduction to Languages and the Theory of Computation (third edition), by John Martin, McGraw-Hill, 2003.
- Introduction to Automata Theory, Languages, and Computation (third edition), by John Hopcroft, Rajeev Motwani, Jeffrey Ullman, Addison Wesley, 2007.

Please let us know if you find errors, typos, simpler proofs, comments, omissions, or if you think that some parts of the book "need improvement".

Chapter 1

Introduction

1.1 Purpose and motivation

This course is on the *Theory of Computation*, which tries to answer the following questions:

- What are the mathematical properties of computer hardware and software?
- What is a *computation* and what is an *algorithm*? Can we give rigorous mathematical definitions of these notions?
- What are the *limitations* of computers? Can "everything" be computed? (As we will see, the answer to this question is "no".)

Purpose of the Theory of Computation: Develop formal mathematical models of computation that reflect real-world computers.

This field of research was started by mathematicians and logicians in the 1930's, when they were trying to understand the meaning of a "computation". A central question asked was whether all mathematical problems can be solved in a systematic way. The research that started in those days led to computers as we know them today.

Nowadays, the Theory of Computation can be divided into the following three areas: Complexity Theory, Computability Theory, and Automata Theory.

1.1.1 Complexity theory

The main question asked in this area is "What makes some problems computationally *hard* and other problems *easy*?"

Informally, a problem is called "easy", if it is efficiently solvable. Examples of "easy" problems are (i) sorting a sequence of, say, 1,000,000 numbers, (ii) searching for a name in a telephone directory, and (iii) computing the fastest way to drive from Ottawa to Miami. On the other hand, a problem is called "hard", if it cannot be solved efficiently, or if we don't know whether it can be solved efficiently. Examples of "hard" problems are (i) time table scheduling for all courses at Carleton, (ii) factoring a 300-digit integer into its prime factors, and (iii) computing a layout for chips in VLSI.

Central Question in Complexity Theory: Classify problems according to their degree of "difficulty". Give a rigorous proof that problems that seem to be "hard" are really "hard".

1.1.2 Computability theory

In the 1930's, Gödel, Turing, and Church discovered that some of the fundamental mathematical problems cannot be solved by a "computer". (This may sound strange, because computers were invented only in the 1940's). An example of such a problem is "Is an arbitrary mathematical statement true or false?" To attack such a problem, we need formal definitions of the notions of

- computer,
- algorithm, and
- computation.

The theoretical models that were proposed in order to understand solvable and unsolvable problems led to the development of real computers.

Central Question in Computability Theory: Classify problems as being solvable or unsolvable.

1.1.3 Automata theory

Automata Theory deals with definitions and properties of different types of "computation models". Examples of such models are:

- Finite Automata. These are used in text processing, compilers, and hardware design.
- Context-Free Grammars. These are used to define programming languages and in Artificial Intelligence.
- Turing Machines. These form a simple abstract model of a "real" computer, such as your PC at home.

Central Question in Automata Theory: Do these models have the same power, or can one model solve more problems than the other?

1.1.4 This course

In this course, we will study the last two areas in reverse order: We will start with Automata Theory, followed by Computability Theory. The first area, Complexity Theory, will be covered in COMP 3804.

Actually, before we start, we will review some mathematical proof techniques. As you may guess, this is a fairly theoretical course, with lots of definitions, theorems, and proofs. You may guess this course is fun stuff for math lovers, but boring and irrelevant for others. You guessed it *wrong*, and here are the reasons:

1. This course is about the fundamental capabilities and limitations of computers. These topics form the core of computer science.
2. It is about mathematical properties of computer hardware and software.
3. This theory is very much relevant to practice, for example, in the design of new programming languages, compilers, string searching, pattern matching, computer security, artificial intelligence, etc., etc.
4. This course helps you to learn problem solving skills. Theory teaches you how to think, prove, argue, solve problems, express, and abstract.

5. This theory simplifies the complex computers to an abstract and simple mathematical model, and helps you to understand them better.

6. This course is about rigorously analyzing capabilities and limitations of systems.

Where does this course fit in the Computer Science Curriculum at Carleton University? It is a theory course that is the third part in the series COMP 1805, COMP 2804, COMP 3803, COMP 3804, and COMP 4804. This course also widens your understanding of computers and will influence other courses including Compilers, Programming Languages, and Artificial Intelligence.

1.2 Mathematical preliminaries

Throughout this course, we will assume that you know the following mathematical concepts:

1. A *set* is a collection of well-defined objects. Examples are (i) the set of all Dutch Olympic Gold Medallists, (ii) the set of all pubs in Ottawa, and (iii) the set of all even natural numbers.

2. The set of *natural numbers* is $\mathbb{N} = \{1, 2, 3, \ldots\}$.

3. The set of *integers* is $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.

4. The set of *rational numbers* is $\mathbb{Q} = \{m/n : m \in \mathbb{Z}, n \in \mathbb{Z}, n \neq 0\}$.

5. The set of *real numbers* is denoted by $\mathbb{R}$.

6. If A and B are sets, then A is a *subset* of B, written as $A \subseteq B$, if every element of A is also an element of B. For example, the set of even natural numbers is a subset of the set of all natural numbers. Every set A is a subset of itself, i.e., $A \subseteq A$. The empty set is a subset of every set A, i.e., $\emptyset \subseteq A$.

7. If B is a set, then the *power set* $\mathcal{P}(B)$ of B is defined to be the set of all subsets of B:
$$\mathcal{P}(B) = \{A : A \subseteq B\}.$$
Observe that $\emptyset \in \mathcal{P}(B)$ and $B \in \mathcal{P}(B)$.

8. If A and B are two sets, then

   (a) their *union* is defined as

   $$A \cup B = \{x : x \in A \text{ or } x \in B\},$$

   (b) their *intersection* is defined as

   $$A \cap B = \{x : x \in A \text{ and } x \in B\},$$

   (c) their *difference* is defined as

   $$A \setminus B = \{x : x \in A \text{ and } x \notin B\},$$

   (d) the *Cartesian product* of A and B is defined as

   $$A \times B = \{(x, y) : x \in A \text{ and } y \in B\},$$

   (e) the *complement* of A is defined as

   $$\overline{A} = \{x : x \notin A\}.$$

9. A *binary relation* on two sets A and B is a subset of $A \times B$.

10. A *function* f from A to B, denoted by $f : A \to B$, is a binary relation R, having the property that for each element $a \in A$, there is exactly one ordered pair in R, whose first component is a. We will also say that $f(a) = b$, or f maps a to b, or the image of a under f is b. The set A is called the *domain* of f, and the set

    $$\{b \in B : \text{ there is an } a \in A \text{ with } f(a) = b\}$$

    is called the *range* of f.

11. A function $f : A \to B$ is *one-to-one* (or *injective*), if for any two distinct elements a and a' in A, we have $f(a) \neq f(a')$. The function f is *onto* (or *surjective*), if for each element $b \in B$, there exists an element $a \in A$, such that $f(a) = b$; in other words, the range of f is equal to the set B. A function f is a *bijection*, if f is both injective and surjective.

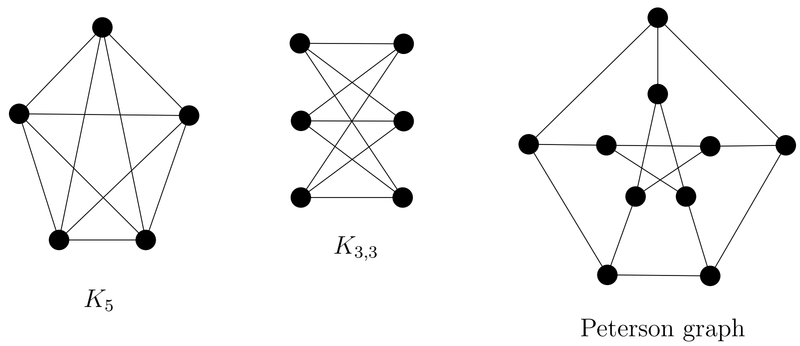
12. A binary relation $R \subseteq A \times A$ is an *equivalence relation*, if it satisfies the following three conditions:

(a) R is *reflexive*: For every element in $a \in A$, we have $(a, a) \in R$.

(b) R is *symmetric*: For all a and b in A, if $(a, b) \in R$, then also $(b, a) \in R$.

(c) R is *transitive*: For all a, b, and c in A, if $(a, b) \in R$ and $(b, c) \in R$, then also $(a, c) \in R$.

13. A *graph* $G = (V, E)$ is a pair consisting of a set V, whose elements are called *vertices*, and a set E, where each element of E is a pair of distinct vertices. The elements of E are called *edges*. The figure below shows some well-known graphs: K_5 (the complete graph on five vertices), $K_{3,3}$ (the complete bipartite graph on $2 \times 3 = 6$ vertices), and the Peterson graph.



The *degree* of a vertex v, denoted by $deg(v)$, is defined to be the number of edges that are incident on v.

A *path* in a graph is a sequence of vertices that are connected by edges. A path is a *cycle*, if it starts and ends at the same vertex. A *simple path* is a path without any repeated vertices. A graph is *connected*, if there is a path between every pair of vertices.

14. In the context of strings, an *alphabet* is a finite set, whose elements are called *symbols*. Examples of alphabets are $\Sigma = \{0, 1\}$ and $\Sigma = \{a, b, c, \ldots, z\}$.

15. A *string* over an alphabet Σ is a finite sequence of symbols, where each symbol is an element of Σ. The *length* of a string w, denoted by $|w|$, is the number of symbols contained in w. The *empty string*, denoted by

ϵ, is the string having length zero. For example, if the alphabet Σ is equal to $\{0, 1\}$, then 10, 1000, 0, 101, and ϵ are strings over Σ, having lengths 2, 4, 1, 3, and 0, respectively.

16. A *language* is a set of strings.

17. The *Boolean values* are 1 and 0, that represent *true* and *false*, respectively. The basic Boolean operations include

    (a) negation (or *NOT*), represented by $\neg$,

    (b) conjunction (or *AND*), represented by $\wedge$,

    (c) disjunction (or *OR*), represented by $\vee$,

    (d) exclusive-or (or *XOR*), represented by $\oplus$,

    (e) equivalence, represented by $\leftrightarrow$ or $\Leftrightarrow$,

    (f) implication, represented by $\rightarrow$ or $\Rightarrow$.

    The following table explains the meanings of these operations.

NOT	*AND*	*OR*	*XOR*	equivalence	implication
$\neg 0 = 1$	$0 \wedge 0 = 0$	$0 \vee 0 = 0$	$0 \oplus 0 = 0$	$0 \leftrightarrow 0 = 1$	$0 \rightarrow 0 = 1$
$\neg 1 = 0$	$0 \wedge 1 = 0$	$0 \vee 1 = 1$	$0 \oplus 1 = 1$	$0 \leftrightarrow 1 = 0$	$0 \rightarrow 1 = 1$
	$1 \wedge 0 = 0$	$1 \vee 0 = 1$	$1 \oplus 0 = 1$	$1 \leftrightarrow 0 = 0$	$1 \rightarrow 0 = 0$
	$1 \wedge 1 = 1$	$1 \vee 1 = 1$	$1 \oplus 1 = 0$	$1 \leftrightarrow 1 = 1$	$1 \rightarrow 1 = 1$

1.3 Proof techniques

In mathematics, a theorem is a statement that is true. A proof is a sequence of mathematical statements that form an argument to show that a theorem is true. The statements in the proof of a theorem include axioms (assumptions about the underlying mathematical structures), hypotheses of the theorem to be proved, and previously proved theorems. The main question is "How do we go about proving theorems?" This question is similar to the question of how to solve a given problem. Of course, the answer is that finding proofs, or solving problems, is not easy; otherwise life would be dull! There is no specified way of coming up with a proof, but there are some generic strategies that could be of help. In this section, we review some of these strategies, that will be sufficient for this course. The best way to get a feeling of how to come up with a proof is by solving a large number of problems. Here are some useful tips. (You may take a look at the book *How to Solve It*, by G. Pólya).

1. Read and completely understand the statement of the theorem to be proved. Most often this is the hardest part.

2. Sometimes, theorems contain theorems inside them. For example, "Property A if and only if property B", requires showing two statements:

   (a) If property A is true, then property B is true ($A \Rightarrow B$).

   (b) If property B is true, then property A is true ($B \Rightarrow A$).

   Another example is the theorem "Set A equals set B." To prove this, we need to prove that $A \subseteq B$ and $B \subseteq A$. That is, we need to show that each element of set A is in set B, and that each element of set B is in set A.

3. Try to work out a few simple cases of the theorem just to get a grip on it (i.e., crack a few simple cases first).

4. Try to write down the proof once you have it. This is to ensure the correctness of your proof. Often, mistakes are found at the time of writing.

5. Finding proofs takes time, we do not come prewired to produce proofs. Be patient, think, express and write clearly and try to be precise as much as possible.

In the next sections, we will go through some of the proof strategies.

1.3.1 Direct proofs

As the name suggests, in a direct proof of a theorem, we just approach the theorem directly.

Theorem 1.3.1 *If n is an odd positive integer, then n^2 is odd as well.*

Proof. An odd positive integer n can be written as $n = 2k + 1$, for some integer $k \geq 0$. Then

$$n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1.$$

Since $2(2k^2 + 2k)$ is even, and "even plus one is odd", we can conclude that n^2 is odd. ∎

Theorem 1.3.2 *Let $G = (V, E)$ be a graph. Then the sum of the degrees of all vertices is an even integer, i.e.,*

$$\sum_{v \in V} deg(v)$$

is even.

Proof. If you do not see the meaning of this statement, then first try it out for a few graphs. The reason why the statement holds is very simple: Each edge contributes 2 to the summation (because an edge is incident on exactly two distinct vertices). ∎

Actually, the proof above proves the following theorem.

Theorem 1.3.3 *Let $G = (V, E)$ be a graph. Then the sum of the degrees of all vertices is equal to twice the number of edges, i.e.,*

$$\sum_{v \in V} deg(v) = 2|E|.$$

1.3.2 Constructive proofs

This technique not only shows the existence of a certain object, it actually gives a method of creating it. Here is how a constructive proof looks like:

Theorem 1.3.4 *There exists an object with property $\mathcal{P}$.*

Proof. Here is the object: [. . .]
And here is the proof that the object satisfies property $\mathcal{P}$: [. . .] ∎

Here is an example of a constructive proof. A graph is called 3-*regular*, if each vertex has degree three.

Theorem 1.3.5 *For every even integer $n \geq 4$, there exists a 3-regular graph with n vertices.*

Proof. Define

$$V = \{0, 1, 2, \ldots, n-1\},$$

and

$$E = \{\{i, i+1\} : 0 \leq i \leq n-2\} \cup \{\{n-1, 0\}\} \cup \{\{i, i+n/2\} : 0 \leq i \leq n/2-1\}.$$

Then the graph $G = (V, E)$ is 3-regular.

Convince yourself that this graph is indeed 3-regular. It may help to draw the graph for, say, $n = 8$. ∎

1.3.3 Nonconstructive proofs

In a nonconstructive proof, we show that a certain object exists, without actually creating it. Here is an example of such a proof:

Theorem 1.3.6 *There exist irrational numbers x and y such that x^y is rational.*

Proof. There are two possible cases.

Case 1: $\sqrt{2}^{\sqrt{2}} \in \mathbb{Q}$.

In this case, we take $x = y = \sqrt{2}$. In Theorem 1.3.9 below, we will prove that $\sqrt{2}$ is irrational.

Case 2: $\sqrt{2}^{\sqrt{2}} \notin \mathbb{Q}$.

In this case, we take $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$. Since

$$x^y = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^2 = 2,$$

the claim in the theorem follows. ∎

Observe that this proof indeed proves the theorem, but it does not give an example of a pair of irrational numbers x and y such that x^y is rational.

1.3.4 Proofs by contradiction

This is how a proof by contradiction looks like:

Theorem 1.3.7 *Statement $\mathcal{S}$ is true.*

Proof. Assume that statement $\mathcal{S}$ is false. Then, derive a contradiction (such as $1 + 1 = 3$).

In other words, show that the statement "$\neg\mathcal{S} \Rightarrow$ *false*" is true. This is sufficient, because the contrapositive of the statement "$\neg\mathcal{S} \Rightarrow$ *false*" is the statement "*true* $\Rightarrow \mathcal{S}$". The latter logical formula is equivalent to $\mathcal{S}$, and that is what we wanted to show. ∎

Below, we give two examples of proofs by contradiction.

Theorem 1.3.8 *Let n be a positive integer. If n^2 is even, then n is even.*

Proof. We will prove the theorem by contradiction. So we assume that n^2 is even, but n is odd. Since n is odd, we know from Theorem 1.3.1 that n^2 is odd. This is a contradiction, because we assumed that n^2 is even. ∎

Theorem 1.3.9 *$\sqrt{2}$ is irrational, i.e., $\sqrt{2}$ cannot be written as a fraction of two integers m and n.*

Proof. We will prove the theorem by contradiction. So we assume that $\sqrt{2}$ is rational. Then $\sqrt{2}$ can be written as a fraction of two integers, $\sqrt{2} = m/n$, where $m \geq 1$ and $n \geq 1$. We may assume that m and n do not share any common factors, i.e., the greatest common divisor of m and n is equal to one; if this is not the case, then we can get rid of the common factors. By squaring $\sqrt{2} = m/n$, we get $2n^2 = m^2$. This implies that m^2 is even. Then, by Theorem 1.3.8, m is even, which means that we can write m as $m = 2k$, for some positive integer k. It follows that $2n^2 = m^2 = 4k^2$, which implies that $n^2 = 2k^2$. Hence, n^2 is even. Again by Theorem 1.3.8, it follows that n is even.

We have shown that m and n are both even. But we know that m and n are *not* both even. Hence, we have a contradiction. Our assumption that $\sqrt{2}$ is rational is wrong. Thus, we can conclude that $\sqrt{2}$ is irrational. ∎

There is a nice discussion of this proof in the book *My Brain is Open: The Mathematical Journeys of Paul Erdős* by B. Schechter.

1.3.5 The pigeon hole principle

This is a simple principle with surprising consequences.

Pigeon Hole Principle: If $n+1$ or more objects are placed into n boxes, then there is at least one box containing two or more objects. In other words, if A and B are two sets such that $|A| > |B|$, then there is no one-to-one function from A to B.

Theorem 1.3.10 *Let n be a positive integer. Every sequence of n^2+1 distinct real numbers contains a subsequence of length $n+1$ that is either increasing or decreasing.*

Proof. For example consider the sequence $(20, 10, 9, 7, 11, 2, 21, 1, 20, 31)$ of $10 = 3^2 + 1$ numbers. This sequence contains an increasing subsequence of length $4 = 3 + 1$, namely $(10, 11, 21, 31)$.

The proof of this theorem is by contradiction, and uses the pigeon hole principle.

Let $(a_1, a_2, \ldots, a_{n^2+1})$ be an arbitrary sequence of $n^2 + 1$ distinct real numbers. For each i with $1 \leq i \leq n^2 + 1$, let inc_i denote the length of the longest increasing subsequence that starts at a_i, and let dec_i denote the length of the longest decreasing subsequence that starts at a_i.

Using this notation, the claim in the theorem can be formulated as follows: There is an index i such that $inc_i \geq n+1$ or $dec_i \geq n+1$.

We will prove the claim by contradiction. So we assume that $inc_i \leq n$ and $dec_i \leq n$ for all i with $1 \leq i \leq n^2 + 1$.

Consider the set

$$B = \{(b, c) : 1 \leq b \leq n, 1 \leq c \leq n\},$$

and think of the elements of B as being boxes. For each i with $1 \leq i \leq n^2+1$, the pair (inc_i, dec_i) is an element of B. So we have n^2+1 elements (inc_i, dec_i), which are placed in the n^2 boxes of B. By the pigeon hole principle, there must be a box that contains two (or more) elements. In other words, there exist two integers i and j such that $i < j$ and

$$(inc_i, dec_i) = (inc_j, dec_j).$$

Recall that the elements in the sequence are distinct. Hence, $a_i \neq a_j$. We consider two cases.

First assume that $a_i < a_j$. Then the length of the longest increasing subsequence starting at a_i must be at least $1 + inc_j$, because we can append a_i to the longest increasing subsequence starting at a_j. Therefore, $inc_i \neq inc_j$, which is a contradiction.

The second case is when $a_i > a_j$. Then the length of the longest decreasing subsequence starting at a_i must be at least $1 + dec_j$, because we can append a_i to the longest decreasing subsequence starting at a_j. Therefore, $dec_i \neq dec_j$, which is again a contradiction. ∎

1.3.6 Proofs by induction

This is a very powerful and important technique for proving theorems.

For each positive integer n, let $P(n)$ be a mathematical statement that depends on n. Assume we wish to prove that $P(n)$ is true for all positive integers n. A proof by induction of such a statement is carried out as follows:

Basis: Prove that $P(1)$ is true.

Induction step: Prove that for all $n \geq 1$, the following holds: If $P(n)$ is true, then $P(n+1)$ is also true.

In the induction step, we choose an arbitrary integer $n \geq 1$ and assume that $P(n)$ is true; this is called the *induction hypothesis*. Then we prove that $P(n+1)$ is also true.

Theorem 1.3.11 *For all positive integers n, we have*

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}.$$

Proof. We start with the basis of the induction. If $n = 1$, then the left-hand side is equal to 1, and so is the right-hand side. So the theorem is true for $n = 1$.

For the induction step, let $n \geq 1$ and assume that the theorem is true for n, i.e., assume that

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}.$$

We have to prove that the theorem is true for $n+1$, i.e., we have to prove that

$$1+2+3+\ldots+(n+1)=\frac{(n+1)(n+2)}{2}.$$

Here is the proof:

$$\begin{aligned}1+2+3+\ldots+(n+1) &= \underbrace{1+2+3+\ldots+n}_{=\frac{n(n+1)}{2}}+(n+1)\\ &= \frac{n(n+1)}{2}+(n+1)\\ &= \frac{(n+1)(n+2)}{2}.\end{aligned}$$

■

By the way, here is an alternative proof of the theorem above: Let $S = 1+2+3+\ldots+n$. Then,

$$\begin{array}{ccccccccccccc} S & = & 1 & + & 2 & + & 3 & + & \ldots & + & (n-2) & + & (n-1) & + & n \\ S & = & n & + & (n-1) & + & (n-2) & + & \ldots & + & 3 & + & 2 & + & 1 \\ \hline 2S & = & (n+1) & + & (n+1) & + & (n+1) & + & \ldots & + & (n+1) & + & (n+1) & + & (n+1) \end{array}$$

Since there are n terms on the right-hand side, we have $2S = n(n+1)$. This implies that $S = n(n+1)/2$.

Theorem 1.3.12 *For every positive integer n, $a-b$ is a factor of $a^n - b^n$.*

Proof. A direct proof can be given by providing a factorization of $a^n - b^n$:

$$a^n - b^n = (a-b)(a^{n-1}+a^{n-2}b+a^{n-3}b^2+\ldots+ab^{n-2}+b^{n-1}).$$

We now prove the theorem by induction. For the basis, let $n=1$. The claim in the theorem is "$a-b$ is a factor of $a-b$", which is obviously true.

Let $n \geq 1$ and assume that $a-b$ is a factor of $a^n - b^n$. We have to prove that $a-b$ is a factor of $a^{n+1} - b^{n+1}$. We have

$$a^{n+1} - b^{n+1} = a^{n+1} - a^n b + a^n b - b^{n+1} = a^n(a-b) + (a^n - b^n)b.$$

The first term on the right-hand side is divisible by $a-b$. By the induction hypothesis, the second term on the right-hand side is divisible by $a-b$ as well. Therefore, the entire right-hand side is divisible by $a-b$. Since the right-hand side is equal to $a^{n+1} - b^{n+1}$, it follows that $a-b$ is a factor of $a^{n+1} - b^{n+1}$. ■

We now give an alternative proof of Theorem 1.3.3:

Theorem 1.3.13 *Let $G = (V, E)$ be a graph with m edges. Then the sum of the degrees of all vertices is equal to twice the number of edges, i.e.,*

$$\sum_{v \in V} deg(v) = 2m.$$

Proof. The proof is by induction on the number m of edges. For the basis of the induction, assume that $m = 0$. Then the graph G does not contain any edges and, therefore, $\sum_{v \in V} deg(v) = 0$. Thus, the theorem is true if $m = 0$.

Let $m \geq 0$ and assume that the theorem is true for every graph with m edges. Let G be an arbitrary graph with $m + 1$ edges. We have to prove that $\sum_{v \in V} deg(v) = 2(m + 1)$.

Let $\{a, b\}$ be an arbitrary edge in G, and let G' be the graph obtained from G by removing the edge $\{a, b\}$. Since G' has m edges, we know from the induction hypothesis that the sum of the degrees of all vertices in G' is equal to $2m$. Using this, we obtain

$$\sum_{v \in G} deg(v) = \sum_{v \in G'} deg(v) + 2 = 2m + 2 = 2(m + 1).$$

∎

1.3.7 More examples of proofs

Recall Theorem 1.3.5, which states that for every *even* integer $n \geq 4$, there exists a 3-regular graph with n vertices. The following theorem explains why we stated this theorem for even values of n.

Theorem 1.3.14 *Let $n \geq 5$ be an odd integer. There is no 3-regular graph with n vertices.*

Proof. The proof is by contradiction. So we assume that there exists a graph $G = (V, E)$ with n vertices that is 3-regular. Let m be the number of edges in G. Since $deg(v) = 3$ for every vertex, we have

$$\sum_{v \in V} deg(v) = 3n.$$

On the other hand, by Theorem 1.3.3, we have

$$\sum_{v \in V} deg(v) = 2m.$$

It follows that $3n = 2m$, which can be rewritten as $m = 3n/2$. Since m is an integer, and since $gcd(2, 3) = 1$, $n/2$ must be an integer. Hence, n is even, which is a contradiction. ∎

Let K_n be the *complete graph* on n vertices. This graph has a vertex set of size n, and every pair of distinct vertices is joined by an edge.

If $G = (V, E)$ is a graph with n vertices, then the *complement* $\overline{G}$ of G is the graph with vertex set V that consists of those edges of K_n that are not present in G.

Theorem 1.3.15 *Let $n \geq 2$ and let G be a graph on n vertices. Then G is connected or $\overline{G}$ is connected.*

Proof. We prove the theorem by induction on the number n of vertices. For the basis, assume that $n = 2$. There are two possibilities for the graph G:

1. G contains one edge. In this case, G is connected.

2. G does not contain an edge. In this case, the complement $\overline{G}$ contains one edge and, therefore, $\overline{G}$ is connected.

So for $n = 2$, the theorem is true.

Let $n \geq 2$ and assume that the theorem is true for every graph with n vertices. Let G be graph with $n + 1$ vertices. We have to prove that G is connected or $\overline{G}$ is connected. We consider three cases.

Case 1: There is a vertex v whose degree in G is equal to n.

Since G has $n+1$ vertices, v is connected by an edge to every other vertex of G. Therefore, G is connected.

Case 2: There is a vertex v whose degree in G is equal to 0.

In this case, the degree of v in the graph $\overline{G}$ is equal to n. Since $\overline{G}$ has $n+1$ vertices, v is connected by an edge to every other vertex of $\overline{G}$. Therefore, $\overline{G}$ is connected.

Case 3: For every vertex v, the degree of v in G is in $\{1, 2, \ldots, n-1\}$.

Let v be an arbitrary vertex of G. Let G' be the graph obtained by deleting from G the vertex v, together with all edges that are incident on v. Since G' has n vertices, we know from the induction hypothesis that G' is connected or $\overline{G'}$ is connected.

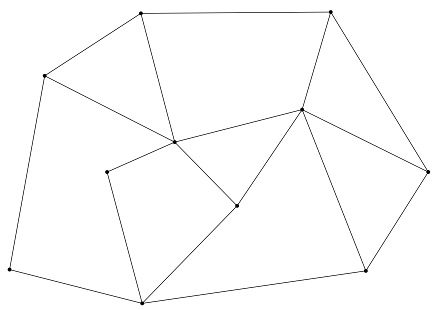
Let us first assume that G' is connected. Then the graph G is connected as well, because there is at least one edge in G between v and some vertex of G'.

If G' is not connected, then $\overline{G'}$ must be connected. Since we are in Case 3, we know that the degree of v in G is in the set $\{1, 2, \ldots, n-1\}$. It follows that the degree of v in the graph $\overline{G}$ is in this set as well. Hence, there is at least one edge in $\overline{G}$ between v and some vertex in $\overline{G'}$. This implies that $\overline{G}$ is connected. ∎

The previous theorem can be rephrased as follows:

Theorem 1.3.16 *Let $n \geq 2$ and consider the complete graph K_n on n vertices. Color each edge of this graph as either red or blue. Let R be the graph consisting of all the red edges, and let B be the graph consisting of all the blue edges. Then R is connected or B is connected.*

A graph is said to be *planar*, if it can be drawn (a better term is "embedded") in the plane in such a way that no two edges intersect, except possibly at their endpoints. An embedding of a planar graph consists of vertices, edges, and faces. In the example below, there are 11 vertices, 18 edges, and 9 faces (including the unbounded face).



The following theorem is known as *Euler's theorem for planar graphs.* Apparently, this theorem was discovered by Euler around 1750. Legendre gave the first proof in 1794, see

`http://www.ics.uci.edu/~eppstein/junkyard/euler/`

Theorem 1.3.17 (Euler) *Consider an embedding of a planar graph G. Let v, e, and f be the number of vertices, edges, and faces (including the single* *unbounded face) of this embedding, respectively. Moreover, let c be the number of connected components of G. Then*

$$v - e + f = c + 1.$$

Proof. The proof is by induction on the number of edges of G. To be more precise, we start with a graph having no edges, and prove that the theorem holds for this case. Then, we add the edges one by one, and show that the relation $v - e + f = c + 1$ is maintained.

So we first assume that G has no edges, i.e., $e = 0$. Then the embedding consists of a collection of v points. In this case, we have $f = 1$ and $c = v$. Hence, the relation $v - e + f = c + 1$ holds.

Let $e > 0$ and assume that Euler's formula holds for a subgraph of G having $e - 1$ edges. Let $\{u, v\}$ be an edge of G that is not in the subgraph, and add this edge to the subgraph. There are two cases depending on whether this new edge joins two connected components or joins two vertices in the same connected component.

Case 1: The new edge $\{u, v\}$ joins two connected components.

In this case, the number of vertices and the number of faces do not change, the number of connected components goes down by 1, and the number of edges increases by 1. It follows that the relation in the theorem is still valid.

Case 2: The new edge $\{u, v\}$ joins two vertices in the same connected component.

In this case, the number of vertices and the number of connected components do not change, the number of edges increases by 1, and the number of faces increases by 1 (because the new edge splits one face into two faces). Therefore, the relation in the theorem is still valid. ∎

Euler's theorem is usually stated as follows:

Theorem 1.3.18 (Euler) *Consider an embedding of a connected planar graph G. Let v, e, and f be the number of vertices, edges, and faces (including the single unbounded face) of this embedding, respectively. Then*

$$v - e + f = 2.$$

If you like surprising proofs of various mathematical results, you should read the book *Proofs from THE BOOK* by Aigner and Ziegler.

Exercises

1.1 Use induction to prove that every integer $n \geq 2$ can be written as a product of prime numbers.

1.2 For every prime number p, prove that $\sqrt{p}$ is irrational.

1.3 Let n be a positive integer that is not a perfect square. Prove that $\sqrt{n}$ is irrational.

1.4 Prove by induction that $n^4 - 4n^2$ is divisible by 3, for all integers $n \geq 1$.

1.5 Prove that

$$\sum_{i=1}^{n} \frac{1}{i^2} < 2 - 1/n,$$

for every integer $n \geq 2$.

1.6 Prove that 9 divides $n^3 + (n+1)^3 + (n+2)^3$, for every integer $n \geq 0$.

1.7 Prove that in any set of $n+1$ numbers from $\{1, 2, \ldots, 2n\}$, there are always two numbers that are consecutive.

1.8 Prove that in any set of $n+1$ numbers from $\{1, 2, \ldots, 2n\}$, there are always two numbers such that one divides the other.

Chapter 2

Finite Automata and Regular Languages

In this chapter, we introduce and analyze the class of languages that are known as *regular languages*. Informally, these languages can be "processed" by computers having a very small amount of memory.

2.1 An example: Controling a toll gate

Before we give a formal definition of a finite automaton, we consider an example in which such an automaton shows up in a natural way. We consider the problem of designing a "computer" that controls a *toll gate*.

When a car arrives at the toll gate, the gate is closed. The gate opens as soon as the driver has payed 25 cents. We assume that we have only three coin denominations: 5, 10, and 25 cents. We also assume that no excess change is returned.

After having arrived at the toll gate, the driver inserts a sequence of coins into the machine. At any moment, the machine has to decide whether or not to open the gate, i.e., whether or not the driver has paid 25 cents (or more). In order to decide this, the machine is in one of the following six *states*, at any moment during the process:

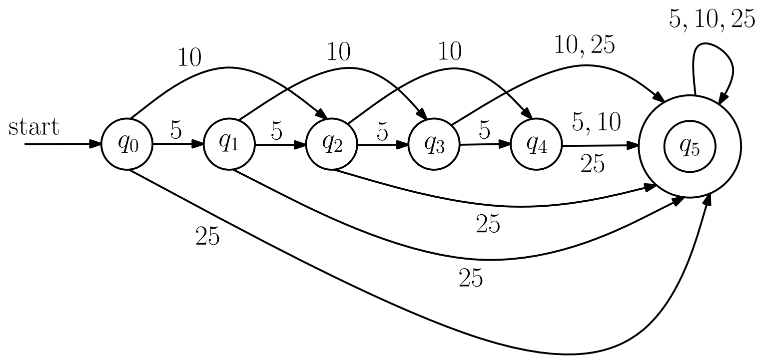
- The machine is in state q_0, if it has not collected any money yet.
- The machine is in state q_1, if it has collected exactly 5 cents.
- The machine is in state q_2, if it has collected exactly 10 cents.

- The machine is in state q_3, if it has collected exactly 15 cents.
- The machine is in state q_4, if it has collected exactly 20 cents.
- The machine is in state q_5, if it has collected 25 cents or more.

Initially (when a car arrives at the toll gate), the machine is in state q_0. Assume, for example, that the driver presents the sequence (10,5,5,10) of coins.

- After receiving the first 10 cents coin, the machine switches from state q_0 to state q_2.
- After receiving the first 5 cents coin, the machine switches from state q_2 to state q_3.
- After receiving the second 5 cents coin, the machine switches from state q_3 to state q_4.
- After receiving the second 10 cents coin, the machine switches from state q_4 to state q_5. At this moment, the gate opens. (Remember that no change is given.)

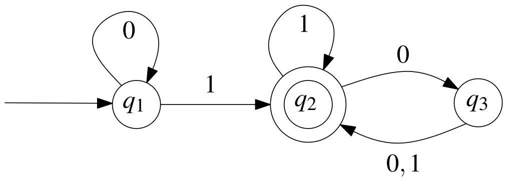
The figure below represents the behavior of the machine for all possible sequences of coins. State q_5 is represented by two circles, because it is a special state: As soon as the machine reaches this state, the gate opens.



Observe that the machine (or computer) only has to remember which state it is in at any given time. Thus, it needs only a very small amount of memory: It has to be able to distinguish between any one of six possible cases and, therefore, it only needs a memory of $\lceil \log 6 \rceil = 3$ bits.

2.2 Deterministic finite automata

Let us look at another example. Consider the following *state diagram*:



We say that q_1 is the start state and q_2 is an accept state. Consider the input string 1101. This string is processed in the following way:

- Initially, the machine is in the start state q_1.
- After having read the first 1, the machine switches from state q_1 to state q_2.
- After having read the second 1, the machine switches from state q_2 to state q_2. (So actually, it does not switch.)
- After having read the first 0, the machine switches from state q_2 to state q_3.
- After having read the third 1, the machine switches from state q_3 to state q_2.

After the entire string 1101 has been processed, the machine is in state q_2, which is an accept state. We say that the string 1101 is accepted by the machine.

Consider now the input string 0101010. After having read this string from left to right (starting in the start state q_1), the machine is in state q_3. Since q_3 is not an accept state, we say that the machine rejects the string 0101010.

We hope you are able to see that this machine accepts every binary string that ends with a 1. In fact, the machine accepts more strings:

- Every binary string having the property that there are an even number of 0s following the rightmost 1, is accepted by this machine.

- Every other binary string is rejected by the machine. Observe that each such string is either empty, consists of 0s only, or has an odd number of 0s following the rightmost 1.

We now come to the formal definition of a finite automaton:

Definition 2.2.1 A *finite automaton* is a 5-tuple $M = (Q, \Sigma, \delta, q, F)$, where

1. Q is a finite set, whose elements are called *states*,
2. Σ is a finite set, called the *alphabet*; the elements of Σ are called *symbols*,
3. $\delta : Q \times \Sigma \to Q$ is a function, called the *transition function*,
4. q is an element of Q; it is called the *start state*,
5. F is a subset of Q; the elements of F are called *accept states*.

You can think of the transition function δ as being the "program" of the finite automaton $M = (Q, \Sigma, \delta, q, F)$. This function tells us what M can do in "one step":

- Let r be a state of Q and let a be a symbol of the alphabet Σ. If the finite automaton M is in state r and reads the symbol a, then it switches from state r to state $\delta(r, a)$. (In fact, $\delta(r, a)$ may be equal to r.)

The "computer" that we designed in the toll gate example in Section 2.1 is a finite automaton. For this example, we have $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, $\Sigma = \{5, 10, 25\}$, the start state is q_0, $F = \{q_5\}$, and δ is given by the following table:

	5	10	25
q_0	q_1	q_2	q_5
q_1	q_2	q_3	q_5
q_2	q_3	q_4	q_5
q_3	q_4	q_5	q_5
q_4	q_5	q_5	q_5
q_5	q_5	q_5	q_5

The example given in the beginning of this section is also a finite automaton. For this example, we have $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, the start state is q_1, $F = \{q_2\}$, and δ is given by the following table:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

Let us denote this finite automaton by M. The language of M, denoted by $L(M)$, is the set of all binary strings that are accepted by M. As we have seen before, we have

$$L(M) = \{w : \ w \text{ contains at least one 1 and ends with an even number of 0s}\}.$$

We now give a formal definition of the language of a finite automaton:

Definition 2.2.2 Let $M = (Q, \Sigma, \delta, q, F)$ be a finite automaton and let $w = w_1 w_2 \ldots w_n$ be a string over Σ. Define the sequence $r_0, r_1, \ldots, r_n$ of states, in the following way:

- $r_0 = q$,
- $r_{i+1} = \delta(r_i, w_{i+1})$, for $i = 0, 1, \ldots, n-1$.

1. If $r_n \in F$, then we say that M *accepts* w.
2. If $r_n \notin F$, then we say that M *rejects* w.

In this definition, w may be the *empty string*, which we denote by ϵ, and whose length is zero; thus in the definition above, $n = 0$. In this case, the sequence $r_0, r_1, \ldots, r_n$ of states has length one; it consists of just the state $r_0 = q$. The empty string is accepted by M if and only if the start state q belongs to F.

Definition 2.2.3 Let $M = (Q, \Sigma, \delta, q, F)$ be a finite automaton. The *language $L(M)$ accepted* by M is defined to be the set of all strings that are accepted by M:

$$L(M) = \{w : \ w \text{ is a string over } \Sigma \text{ and } M \text{ accepts } w\ \}.$$

Definition 2.2.4 A language A is called *regular*, if there exists a finite automaton M such that $A = L(M)$.

We finish this section by presenting an equivalent way of defining the language accepted by a finite automaton. Let $M = (Q, \Sigma, \delta, q, F)$ be a finite automaton. The transition function $\delta : Q \times \Sigma \to Q$ tells us that, when M is in state $r \in Q$ and reads symbol $a \in \Sigma$, it switches from state r to state $\delta(r, a)$. Let Σ^* denote the set of all strings over the alphabet Σ. (Σ^* includes the empty string ϵ.) We extend the function δ to a function

$$\overline{\delta} : Q \times \Sigma^* \to Q,$$

that is defined as follows. For any state $r \in Q$ and for any string w over the alphabet Σ,

$$\overline{\delta}(r, w) = \begin{cases} r & \text{if } w = \epsilon, \\ \delta(\overline{\delta}(r, v), a) & \text{if } w = va, \text{ where } v \text{ is a string and } a \in \Sigma. \end{cases}$$

What is the meaning of this function $\overline{\delta}$? Let r be a state of Q and let w be a string over the alphabet Σ. Then

- $\overline{\delta}(r, w)$ is the state that M reaches, when it starts in state r, reads the string w from left to right, and uses δ to switch from state to state.

Using this notation, we have

$$L(M) = \{w : \ w \text{ is a string over } \Sigma \text{ and } \overline{\delta}(q, w) \in F\}.$$

2.2.1 A first example of a finite automaton

Let

$$A = \{w : \ w \text{ is a binary string containing an odd number of 1s}\}.$$

We claim that this language A is regular. In order to prove this, we have to construct a finite automaton M such that $A = L(M)$.

How to construct M? Here is a first idea: The finite automaton reads the input string w from left to right and keeps track of the number of 1s it has seen. After having read the entire string w, it checks whether this number is odd (in which case w is accepted) or even (in which case w is rejected). Using this approach, the finite automaton needs a state *for every* integer $i \geq 0$, indicating that the number of 1s read so far is equal to i. Hence, to design a finite automaton that follows this approach, we need an *infinite* number of states. But, the definition of finite automaton requires the number of states to be *finite*.

A better, and correct approach, is to keep track of whether the number of 1s read so far is even or odd. This leads to the following finite automaton:

- The set of states is $Q = \{q_e, q_o\}$. If the finite automaton is in state q_e, then it has read an even number of 1s; if it is in state q_o, then it has read an odd number of 1s.

- The alphabet is $\Sigma = \{0, 1\}$.

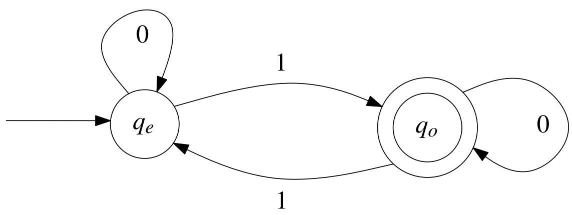
- The start state is q_e, because at the start, the number of 1s read by the automaton is equal to 0, and 0 is even.

- The set F of accept states is $F = \{q_o\}$.

- The transition function δ is given by the following table:

	0	1
q_e	q_e	q_o
q_o	q_o	q_e

This finite automaton $M = (Q, \Sigma, \delta, q_e, F)$ can also be described by its *state diagram*, which is given in the figure below. The arrow that comes "out of the blue" and enters the state q_e, indicates that q_e is the start state. The state depicted with double circles indicates the accept state.



We have constructed a finite automaton M that accepts the language A. Therefore, A is a regular language.

2.2.2 A second example of a finite automaton

Define the language A as

$$A = \{w : w \text{ is a binary string containing 101 as a substring}\}.$$

Again, we claim that A is a regular language. In other words, we claim that there exists a finite automaton M that accepts A, i.e., $A = L(M)$.

The finite automaton M will do the following, when reading an input string from left to right:

- It skips over all 0s, and stays in the start state.
- At the first 1, it switches to the state "maybe the next two symbols are 01".
  - If the next symbol is 1, then it stays in the state "maybe the next two symbols are 01".
  - On the other hand, if the next symbol is 0, then it switches to the state "maybe the next symbol is 1".
    - If the next symbol is indeed 1, then it switches to the accept state (but keeps on reading until the end of the string).
    - On the other hand, if the next symbol is 0, then it switches to the start state, and skips 0s until it reads 1 again.

By defining the following four states, this process will become clear:

- q_1: M is in this state if the last symbol read was 1, but the substring 101 has not been read.
- q_{10}: M is in this state if the last two symbols read were 10, but the substring 101 has not been read.
- q_{101}: M is in this state if the substring 101 has been read in the input string.
- q: In all other cases, M is in this state.

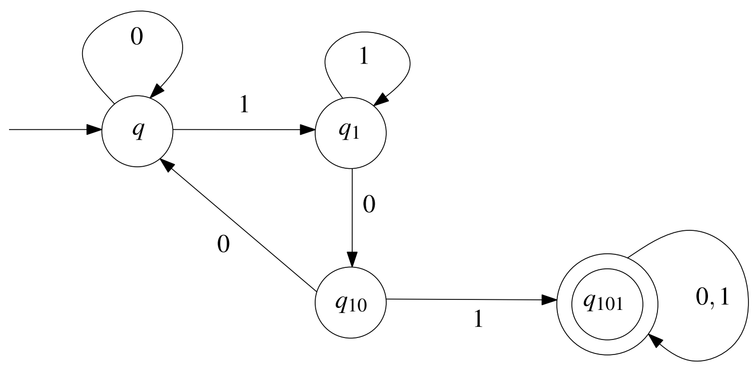
Here is the formal description of the finite automaton that accepts the language A:

- $Q = \{q, q_1, q_{10}, q_{101}\}$,

- $\Sigma = \{0, 1\}$,
- the start state is q,
- the set F of accept states is equal to $F = \{q_{101}\}$, and
- the transition function δ is given by the following table:

	0	1
q	q	q_1
q_1	q_{10}	q_1
q_{10}	q	q_{101}
q_{101}	q_{101}	q_{101}

The figure below gives the state diagram of the finite automaton $M = (Q, \Sigma, \delta, q, F)$.



This finite automaton accepts the language A consisting of all binary strings that contain the substring 101. As an exercise, how would you obtain a finite automaton that accepts the complement of A, i.e., the language consisting of all binary strings that *do not* contain the substring 101?

2.2.3 A third example of a finite automaton

The finite automata we have seen so far have exactly one accept state. In this section, we will see an example of a finite automaton having more accept states.

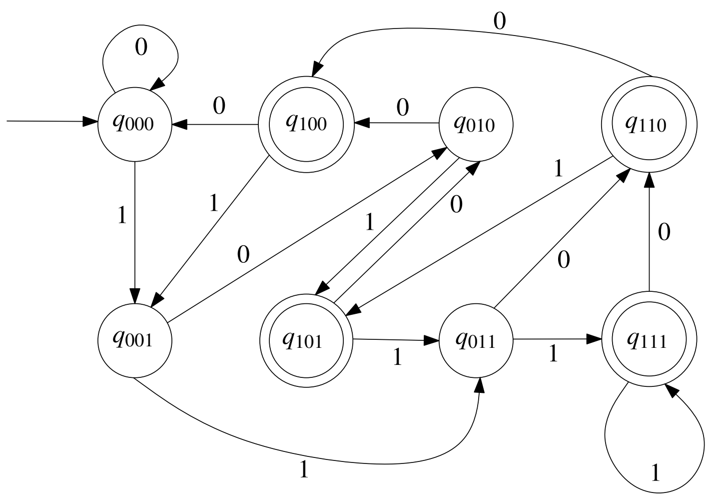
Let A be the language

$$A = \{w \in \{0,1\}^* : \ w \text{ has a 1 in the third position from the right}\},$$

where $\{0,1\}^*$ is the set of all binary strings, including the empty string ϵ. We claim that A is a regular language. To prove this, we have to construct a finite automaton M such that $A = L(M)$. At first sight, it seems difficult (or even impossible?) to construct such a finite automaton: How does the automaton "know" that it has reached the third symbol from the right? It is, however, possible to construct such an automaton. The main idea is to remember the last three symbols that have been read. Thus, the finite automaton has eight states q_{ijk}, where i, j, and k range over the two elements of $\{0,1\}$. If the automaton is in state q_{ijk}, then the following hold:

- If M has read at least three symbols, then the three most recently read symbols are ijk.
- If M has read only two symbols, then these two symbols are jk; moreover, $i = 0$.
- If M has read only one symbol, then this symbol is k; moreover, $i = j = 0$.
- If M has not read any symbol, then $i = j = k = 0$.

The start state is q_{000} and the set of accept states is $\{q_{100}, q_{110}, q_{101}, q_{111}\}$. The transition function of M is given by the following state diagram.



2.3 Regular operations

In this section, we define three operations on languages. Later, we will answer the question whether the set of all regular languages is closed under these operations. Let A and B be two languages over the same alphabet.

1. The *union* of A and B is defined as

$$A \cup B = \{w : \ w \in A \text{ or } w \in B\}.$$

2. The *concatenation* of A and B is defined as

$$AB = \{ww' : \ w \in A \text{ and } w' \in B\}.$$

In words, AB is the set of all strings obtained by taking an arbitrary string w in A and an arbitrary string w' in B, and gluing them together (such that w is to the left of w').

3. The *star* of A is defined as

$$A^* = \{u_1u_2 \ldots u_k : \ k \geq 0 \text{ and } u_i \in A \text{ for all } i = 1, 2, \ldots, k\}.$$

In words, A^* is obtained by taking any finite number of strings in A, and gluing them together. Observe that $k = 0$ is allowed; this corresponds to the empty string ϵ. Thus, $\epsilon \in A^*$.

To give an example, let $A = \{0, 01\}$ and $B = \{1, 10\}$. Then

$$A \cup B = \{0, 01, 1, 10\},$$

$$AB = \{01, 010, 011, 0110\},$$

and

$$A^* = \{\epsilon, 0, 01, 00, 001, 010, 0101, 000, 0001, 00101, \ldots\}.$$

As another example, if $\Sigma = \{0, 1\}$, then Σ^* is the set of all binary strings (including the empty string). Observe that a string always has a finite length.

Before we proceed, we give an alternative (and equivalent) definition of the star of the language A: Define

$$A^0 = \{\epsilon\}$$

and, for $k \geq 1$,

$$A^k = AA^{k-1},$$

i.e., A^k is the concatenation of the two languages A and A^{k-1}. Then we have

$$A^* = \bigcup_{k=0}^{\infty} A^k.$$

Theorem 2.3.1 *The set of regular languages is closed under the union operation, i.e., if A and B are regular languages over the same alphabet Σ, then $A \cup B$ is also a regular language.*

Proof. Since A and B are regular languages, there are finite automata $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ that accept A and B, respectively. In order to prove that $A \cup B$ is regular, we have to construct a finite automaton M that accepts $A \cup B$. In other words, M must have the property that for every string $w \in \Sigma^*$,

$$M \text{ accepts } w \Leftrightarrow M_1 \text{ accepts } w \text{ or } M_2 \text{ accepts } w.$$

As a first idea, we may think that M could do the following:

- Starting in the start state q_1 of M_1, M "runs" M_1 on w.

- If, after having read w, M_1 is in a state of F_1, then $w \in A$, thus $w \in A \cup B$ and, therefore, M accepts w.

- On the other hand, if, after having read w, M_1 is in a state that is not in F_1, then $w \notin A$ and M "runs" M_2 on w, starting in the start state q_2 of M_2. If, after having read w, M_2 is in a state of F_2, then we know that $w \in B$, thus $w \in A \cup B$ and, therefore, M accepts w. Otherwise, we know that $w \notin A \cup B$, and M rejects w.

This idea does not work, because the *finite automaton M can read the input string w only once*. The correct approach is to *run M_1 and M_2 simultaneously*. We define the set Q of states of M to be the Cartesian product $Q_1 \times Q_2$. If M is in state (r_1, r_2), this means that

- if M_1 would have read the input string up to this point, then it would be in state r_1, and

- if M_2 would have read the input string up to this point, then it would be in state r_2.

This leads to the finite automaton $M = (Q, \Sigma, \delta, q, F)$, where

- $Q = Q_1 \times Q_2 = \{(r_1, r_2) : r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$. Observe that $|Q| = |Q_1| \times |Q_2|$, which is finite.
- Σ is the alphabet of A and B (recall that we assume that A and B are languages over the same alphabet).
- The start state q of M is equal to $q = (q_1, q_2)$.
- The set F of accept states of M is given by

$$F = \{(r_1, r_2) : r_1 \in F_1 \text{ or } r_2 \in F_2\} = (F_1 \times Q_2) \cup (Q_1 \times F_2).$$

- The transition function $\delta : Q \times \Sigma \to Q$ is given by

$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)),$$

  for all $r_1 \in Q_1$, $r_2 \in Q_2$, and $a \in \Sigma$.

To finish the proof, we have to show that this finite automaton M indeed accepts the language $A \cup B$. Intuitively, this should be clear from the discussion above. The easiest way to give a formal proof is by using the extended transition functions $\overline{\delta_1}$ and $\overline{\delta_2}$. (The extended transition function has been defined after Definition 2.2.4.) Here we go: Recall that we have to prove that

$$M \text{ accepts } w \Leftrightarrow M_1 \text{ accepts } w \text{ or } M_2 \text{ accepts } w,$$

i.e,

$$M \text{ accepts } w \Leftrightarrow \overline{\delta_1}(q_1, w) \in F_1 \text{ or } \overline{\delta_2}(q_2, w) \in F_2.$$

In terms of the extended transition function $\overline{\delta}$ of the transition function δ of M, this becomes

$$\overline{\delta}((q_1, q_2), w) \in F \Leftrightarrow \overline{\delta_1}(q_1, w) \in F_1 \text{ or } \overline{\delta_2}(q_2, w) \in F_2. \tag{2.1}$$

By applying the definition of the extended transition function, as given after Definition 2.2.4, to δ, it can be seen that

$$\overline{\delta}((q_1, q_2), w) = (\overline{\delta_1}(q_1, w), \overline{\delta_2}(q_2, w)).$$

The latter equality implies that (2.1) is true and, therefore, M indeed accepts the language $A \cup B$. ■

What about the closure of the regular languages under the concatenation and star operations? It turns out that the regular languages are closed under these operations. But how do we prove this?

Let A and B be two regular languages, and let M_1 and M_2 be finite automata that accept A and B, respectively. How do we construct a finite automaton M that accepts the concatenation AB? Given an input string u, M has to decide whether or not u can be broken into two strings w and w' (i.e., write u as $u = ww'$), such that $w \in A$ and $w' \in B$. In words, M has to decide whether or not u can be broken into two substrings, such that the first substring is accepted by M_1 and the second substring is accepted by M_2. The difficulty is caused by the fact that M has to make this decision by scanning the string u only once. If $u \in AB$, then M has to decide, *during this single scan*, where to break u into two substrings. Similarly, if $u \notin AB$, then M has to decide, *during this single scan*, that u *cannot* be broken into two substrings such that the first substring is in A and the second substring is in B.

It seems to be even more difficult to prove that A^* is a regular language, if A itself is regular. In order to prove this, we need a finite automaton that, when given an arbitrary input string u, decides whether or not u can be broken into substrings such that each substring is in A. The problem is that, if $u \in A^*$, the finite automaton has to determine into how many substrings, and where, the string u has to be broken; it has to do this during one single scan of the string u.

As we mentioned already, if A and B are regular languages, then both AB and A^* are also regular. In order to prove these claims, we will introduce a more general type of finite automaton.

The finite automata that we have seen so far are *deterministic*. This means the following:

- If the finite automaton M is in state r and if it reads the symbol a, then M switches from state r to the uniquely defined state $\delta(r, a)$.

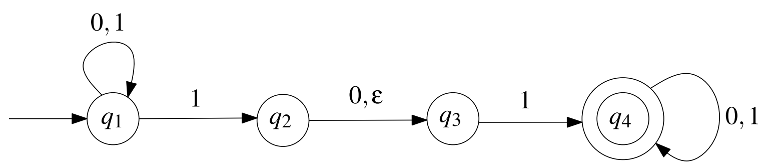
From now on, we will call such a finite automaton a *deterministic finite automaton (DFA)*. In the next section, we will define the notion of a *nondeterministic finite automaton (NFA)*. For such an automaton, there are zero or more possible states to switch to. At first sight, nondeterministic finite automata seem to be more powerful than their deterministic counterparts. We will prove, however, that DFAs have the same power as NFAs. As we will see, using this fact, it will be easy to prove that the class of regular languages is closed under the concatenation and star operations.

2.4 Nondeterministic finite automata

We start by giving three examples of nondeterministic finite automata. These examples will show the difference between this type of automata and the deterministic versions that we have considered in the previous sections. After these examples, we will give a formal definition of a nondeterministic finite automaton.

2.4.1 A first example

Consider the following state diagram:



You will notice three differences with the finite automata that we have seen until now. First, if the automaton is in state q_1 and reads the symbol 1, then it has two options: Either it stays in state q_1, or it switches to state q_2. Second, if the automaton is in state q_2, then it can switch to state q_3 *without reading a symbol*; this is indicated by the edge having the empty string ϵ as label. Third, if the automaton is in state q_3 and reads the symbol 0, then it cannot continue.

Let us see what this automaton can do when it gets the string 010110 as input. Initially, the automaton is in the start state q_1.

- Since the first symbol in the input string is 0, the automaton stays in state q_1 after having read this symbol.

- The second symbol is 1, and the automaton can either stay in state q_1 or switch to state q_2.

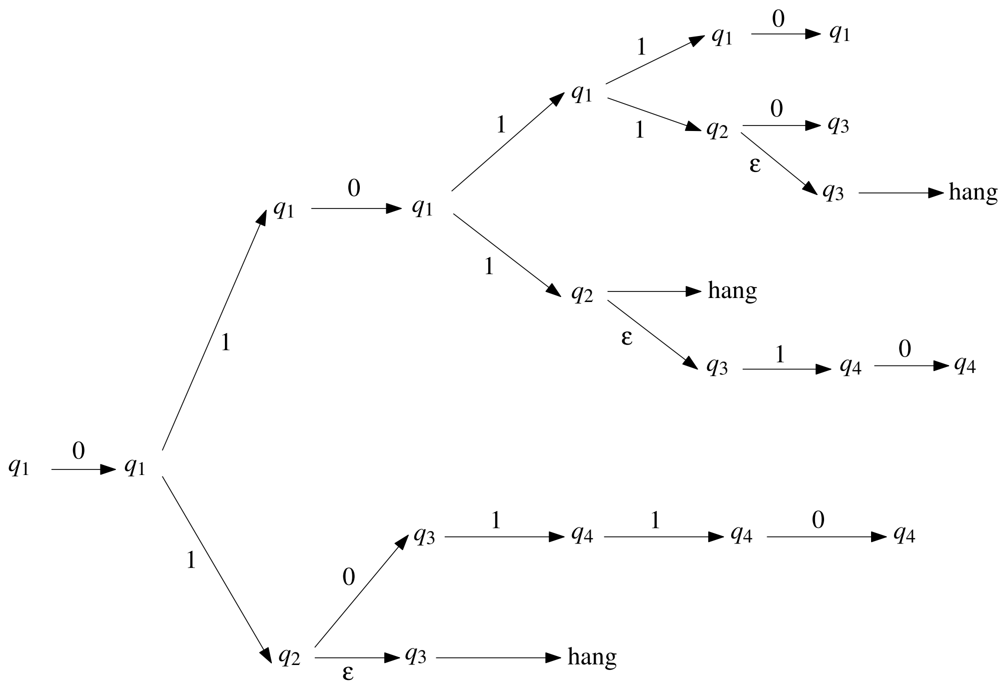
– If the automaton stays in state q_1, then it is still in this state after having read the third symbol.

– If the automaton switches to state q_2, then it again has two options:

  ∗ Either read the third symbol in the input string, which is 0, and switch to state q_3,

  ∗ or switch to state q_3, without reading the third symbol.

If we continue in this way, then we see that, for the input string 010110, there are seven possible computations. All these computations are given in the figure below.



Consider the lowest path in the figure above:

- When reading the first symbol, the automaton stays in state q_1.
- When reading the second symbol, the automaton switches to state q_2.
- The automaton does not read the third symbol (equivalently, it "reads" the empty string ϵ), and switches to state q_3. At this moment, the

automaton cannot continue: The third symbol is 0, but there is no edge leaving q_3 that is labeled 0, and there is no edge leaving q_3 that is labeled ϵ. Therefore, the computation *hangs* at this point.

From the figure, you can see that, out of the seven possible computations, exactly two end in the accept state q_4 (after the entire input string 010110 has been read). We say that the automaton accepts the string 010110, because there is at least one computation that ends in the accept state.

Now consider the input string 010. In this case, there are three possible computations:

1. $q_1 \xrightarrow{0} q_1 \xrightarrow{1} q_1 \xrightarrow{0} q_1$
2. $q_1 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_3$
3. $q_1 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{\epsilon} q_3 \rightarrow \text{hang}$

None of these computations ends in the accept state (after the entire input string 010 has been read). Therefore, we say that the automaton rejects the input string 010.

The state diagram given above is an example of a nondeterministic finite automaton (NFA). Informally, an NFA accepts a string, if there exists *at least one path* in the state diagram that (i) starts in the start state, (ii) does not hang before the entire string has been read, and (iii) ends in an accept state. A string for which (i), (ii), and (iii) does not hold is rejected by the NFA.

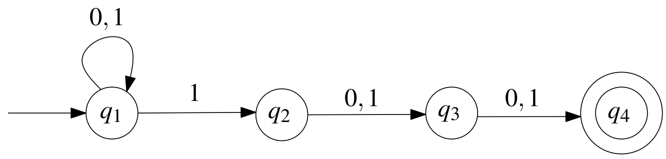
The NFA given above accepts all binary strings that contain 101 or 11 as a substring. All other binary strings are rejected.

2.4.2 A second example

Let A be the language

$$A = \{w \in \{0,1\}^* : \ w \text{ has a 1 in the third position from the right}\}.$$

The following state diagram defines an NFA that accepts all strings that are in A, and rejects all strings that are not in A.



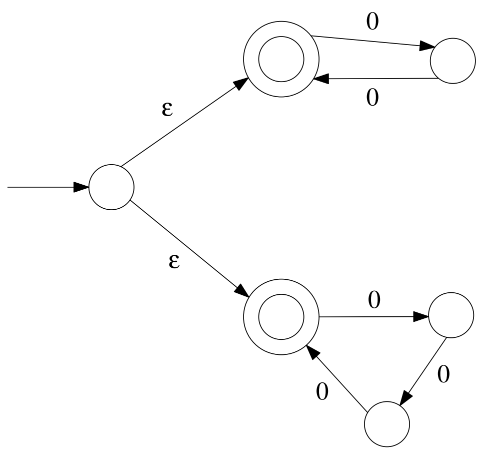
This NFA does the following. If it is in the start state q_1 and reads the symbol 1, then it either stays in state q_1 or it "guesses" that this symbol is the third symbol from the right in the input string. In the latter case, the NFA switches to state q_2, and then it "verifies" that there are indeed exactly two remaining symbols in the input string. If there are more than two remaining symbols, then the NFA hangs (in state q_4) after having read the next two symbols.

Observe how this guessing mechanism is used: The automaton can only read the input string once, from left to right. Hence, it does not know when it reaches the third symbol from the right. When the NFA reads a 1, it can guess that this is the third symbol from the right; after having made this guess, it verifies whether or not the guess was correct.

In Section 2.2.3, we have seen a DFA for the same language A. Observe that the NFA has a much simpler structure than the DFA.

2.4.3 A third example

Consider the following state diagram, which defines an NFA whose alphabet is $\{0\}$.



This NFA accepts the language

$$A = \{0^k : k \equiv 0 \bmod 2 \text{ or } k \equiv 0 \bmod 3\},$$

where 0^k is the string consisting of k many 0s. (If $k = 0$, then $0^k = \epsilon$.) Observe that A is the union of the two languages

$$A_1 = \{0^k : k \equiv 0 \bmod 2\}$$

and

$$A_2 = \{0^k : k \equiv 0 \bmod 3\}.$$

The NFA basically consists of two DFAs: one of these accepts A_1, whereas the other accepts A_2. Given an input string w, the NFA has to decide whether or not $w \in A$, which is equivalent to deciding whether or not $w \in A_1$ or $w \in A_2$. The NFA makes this decision in the following way: At the start, it "guesses" whether (i) it is going to check whether or not $w \in A_1$ (i.e., the length of w is even), or (ii) it is going to check whether or not $w \in A_2$ (i.e., the length of w is a multiple of 3). After having made the guess, it verifies whether or not the guess was correct. If $w \in A$, then there exists a way of making the correct guess and verifying that w is indeed an element of A (by ending in an accept state). If $w \notin A$, then no matter which guess is made, the NFA will never end in an accept state.

2.4.4 Definition of nondeterministic finite automaton

The previous examples give you an idea what nondeterministic finite automata are and how they work. In this section, we give a formal definition of these automata.

For any alphabet Σ, we define Σ_ϵ to be the set

$$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}.$$

Recall the notion of a *power set*: For any set Q, the power set of Q, denoted by $\mathcal{P}(Q)$, is the set of all subsets of Q, i.e.,

$$\mathcal{P}(Q) = \{R : R \subseteq Q\}.$$

Definition 2.4.1 A *nondeterministic finite automaton (NFA)* is a 5-tuple $M = (Q, \Sigma, \delta, q, F)$, where

1. Q is a finite set, whose elements are called *states*,
2. Σ is a finite set, called the *alphabet*; the elements of Σ are called *symbols*,
3. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is a function, called the *transition function*,
4. q is an element of Q; it is called the *start state*,
5. F is a subset of Q; the elements of F are called *accept states*.

As for DFAs, the transition function δ can be thought of as the "program" of the finite automaton $M = (Q, \Sigma, \delta, q, F)$:

- Let $r \in Q$, and let $a \in \Sigma_\epsilon$. Then $\delta(r, a)$ is a (possibly empty) subset of Q. If the NFA M is in state r, and if it reads a (where a may be the empty string ϵ), then M can switch from state r to *any* state in $\delta(r, a)$. If $\delta(r, a) = \emptyset$, then M cannot continue and the computation hangs.

The example given in Section 2.4.1 is an NFA, where $Q = \{q_1, q_2, q_3, q_4\}$, $\Sigma = \{0, 1\}$, the start state is q_1, the set of accept states is $F = \{q_4\}$, and the transition function δ is given by the following table:

	0	1	ϵ
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

Definition 2.4.2 Let $M = (Q, \Sigma, \delta, q, F)$ be an NFA, and let $w \in \Sigma^*$. We say that M *accepts* w, if w can be written as $w = y_1 y_2 \ldots y_m$, where $y_i \in \Sigma_\epsilon$ for all i with $1 \leq i \leq m$, and there *exists* a sequence $r_0, r_1, \ldots, r_m$ of states in Q, such that

- $r_0 = q$,
- $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, 1, \ldots, m-1$, and
- $r_m \in F$.

Otherwise, we say that M *rejects* the string w.

The NFA in the example in Section 2.4.1 accepts the string 01100. This can be seen by taking

- $w = 01\epsilon100 = y_1 y_2 y_3 y_4 y_5 y_6$, and
- $r_0 = q_1$, $r_1 = q_1$, $r_2 = q_2$, $r_3 = q_3$, $r_4 = q_4$, $r_5 = q_4$, and $r_6 = q_4$.

Definition 2.4.3 Let $M = (Q, \Sigma, \delta, q, F)$ be an NFA. The *language* $L(M)$ *accepted* by M is defined as

$$L(M) = \{w \in \Sigma^* : M \text{ accepts } w\}.$$

2.5 Equivalence of DFAs and NFAs

You may have the impression that nondeterministic finite automata are more powerful than deterministic finite automata. In this section, we will show that this is not the case. That is, we will prove that a language can be accepted by a DFA if and only if it can be accepted by an NFA. In order to prove this, we will show how to convert an arbitrary NFA to a DFA that accepts the same language.

What about converting a DFA to an NFA? Well, there is (almost) nothing to do, because a DFA is also an NFA. This is not quite true, because

- the transition function of a DFA maps a state and a symbol to a state, whereas
- the transition function of an NFA maps a state and a symbol to a *set* of zero or more states.

The formal conversion of a DFA to an NFA is done as follows: Let $M = (Q, \Sigma, \delta, q, F)$ be a DFA. Recall that δ is a function $\delta : Q \times \Sigma \rightarrow Q$. We define the function $\delta' : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$ as follows. For any $r \in Q$ and for any $a \in \Sigma_\epsilon$,

$$\delta'(r, a) = \begin{cases} \{\delta(r, a)\} & \text{if } a \neq \epsilon, \\ \emptyset & \text{if } a = \epsilon. \end{cases}$$

Then $N = (Q, \Sigma, \delta', q, F)$ is an NFA, whose behavior is exactly the same as that of the DFA M; the easiest way to see this is by observing that the state diagrams of M and N are equal. Therefore, we have $L(M) = L(N)$.

In the rest of this section, we will show how to convert an NFA to a DFA:

Theorem 2.5.1 *Let $N = (Q, \Sigma, \delta, q, F)$ be a nondeterministic finite automaton. There exists a deterministic finite automaton M, such that $L(M) = L(N)$.*

Proof. Recall that the NFA N can (in general) perform more than one computation on a given input string. The idea of the proof is to construct a DFA M that runs *all these different computations simultaneously.* (We have seen this idea already in the proof of Theorem 2.3.1.) To be more precise, the DFA M will have the following property:

- the state that M is in after having read an initial part of the input string corresponds exactly to the set of all states that N can reach after having read the same part of the input string.

We start by presenting the conversion for the case when N does not contain ϵ-transitions. In other words, the state diagram of N does not contain any edge that has ϵ as a label. (Later, we will extend the conversion to the general case.) Let the DFA M be defined as $M = (Q', \Sigma, \delta', q', F')$, where

- the set Q' of states is equal to $Q' = \mathcal{P}(Q)$; observe that $|Q'| = 2^{|Q|}$,
- the start state q' is equal to $q' = \{q\}$; so M has the "same" start state as N,
- the set F' of accept states is equal to the set of all elements R of Q' having the property that R contains at least one accept state of N, i.e.,

$$F' = \{R \in Q' : R \cap F \neq \emptyset\},$$

- the transition function $\delta' : Q' \times \Sigma \to Q'$ is defined as follows: For each $R \in Q'$ and for each $a \in \Sigma$,

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).$$

Let us see what the transition function δ' of M does. First observe that, since N is an NFA, $\delta(r, a)$ is a subset of Q. This implies that $\delta'(R, a)$ is the union of subsets of Q and, therefore, also a subset of Q. Hence, $\delta'(R, a)$ is an element of Q'.

The set $\delta(r, a)$ is equal to the set of all states of the NFA N that can be reached from state r by reading the symbol a. We take the union of these sets $\delta(r, a)$, where r ranges over all elements of R, to obtain the new set $\delta'(R, a)$. This new set is the state that the DFA M reaches from state R, by reading the symbol a.

In this way, we obtain the correspondence that was given in the beginning of this proof.

After this warming-up, we can consider the general case. In other words, from now on, we allow ϵ-transitions in the NFA N. The DFA M is defined as above, except that the start state q' and the transition function δ' have to be modified. Recall that a computation of the NFA N consists of the following:

1. Start in the start state q and make zero or more ϵ-transitions.
2. Read one "real" symbol of Σ and move to a new state (or stay in the current state).

3. Make zero or more ϵ-transitions.

4. Read one "real" symbol of Σ and move to a new state (or stay in the current state).

5. Make zero or more ϵ-transitions.

6. Etc.

The DFA M will simulate this computation in the following way:

- Simulate 1. in one single step. As we will see below, this simulation is implicitly encoded in the definition of the start state q' of M.
- Simulate 2. and 3. in one single step.
- Simulate 4. and 5. in one single step.
- Etc.

Thus, in *one* step, the DFA M simulates the reading of one "real" symbol of Σ, followed by making zero or more ϵ-transitions.

To formalize this, we need the notion of *ϵ-closure*. For any state r of the NFA N, the ϵ-closure of r, denoted by $C_\epsilon(r)$, is defined to be the set of all states of N that can be reached from r, by making zero or more ϵ-transitions. For any state R of the DFA M (hence, $R \subseteq Q$), we define

$$C_\epsilon(R) = \bigcup_{r \in R} C_\epsilon(r).$$

How do we define the start state q' of the DFA M? Before the NFA N reads its first "real" symbol of Σ, it makes zero or more ϵ-transitions. In other words, at the moment when N reads the first symbol of Σ, it can be in any state of $C_\epsilon(q)$. Therefore, we define q' to be

$$q' = C_\epsilon(q) = C_\epsilon(\{q\}).$$

How do we define the transition function δ' of the DFA M? Assume that M is in state R, and reads the symbol a. At this moment, the NFA N would have been in any state r of R. By reading the symbol a, N can switch to any state in $\delta(r, a)$, and then make zero or more ϵ-transitions. Hence, the

NFA can switch to any state in the set $C_\epsilon(\delta(r,a))$. Based on this, we define $\delta'(R,a)$ to be

$$\delta'(R,a) = \bigcup_{r \in R} C_\epsilon(\delta(r,a)).$$

To summarize, the NFA $N = (Q, \Sigma, \delta, q, F)$ is converted to the DFA $M = (Q', \Sigma, \delta', q', F')$, where

- $Q' = \mathcal{P}(Q)$,
- $q' = C_\epsilon(\{q\})$,
- $F' = \{R \in Q' : R \cap F \neq \emptyset\}$,
- $\delta' : Q' \times \Sigma \to Q'$ is defined as follows: For each $R \in Q'$ and for each $a \in \Sigma$,

$$\delta'(R,a) = \bigcup_{r \in R} C_\epsilon(\delta(r,a)).$$

∎

The results proved until now can be summarized in the following theorem.

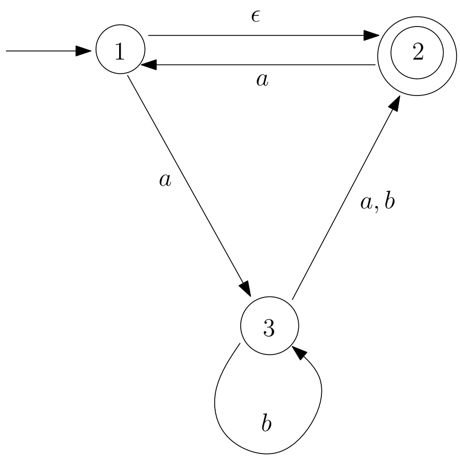
Theorem 2.5.2 *Let A be a language. Then A is regular if and only if there exists a nondeterministic finite automaton that accepts A.*

2.5.1 An example

Consider the NFA $N = (Q, \Sigma, \delta, q, F)$, where $Q = \{1,2,3\}$, $\Sigma = \{a,b\}$, $q = 1$, $F = \{2\}$, and δ is given by the following table:

	a	b	ϵ
1	$\{3\}$	$\emptyset$	$\{2\}$
2	$\{1\}$	$\emptyset$	$\emptyset$
3	$\{2\}$	$\{2,3\}$	$\emptyset$

The state diagram of N is as follows:



We will show how to convert this NFA N to a DFA M that accepts the same language. Following the proof of Theorem 2.5.1, the DFA M is specified by $M = (Q', \Sigma, \delta', q', F')$, where each of the components is defined below.

- $Q' = \mathcal{P}(Q)$. Hence,

$$Q' = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

- $q' = C_\epsilon(\{q\})$. Hence, the start state q' of M is the set of all states of N that can be reached from N's start state $q = 1$, by making zero or more ϵ-transitions. We obtain

$$q' = C_\epsilon(\{q\}) = C_\epsilon(\{1\}) = \{1,2\}.$$

- $F' = \{R \in Q' : R \cap F \neq \emptyset\}$. Hence, the accept states of M are those states that contain the accept state 2 of N. We obtain

$$F' = \{\{2\}, \{1,2\}, \{2,3\}, \{1,2,3\}\}.$$

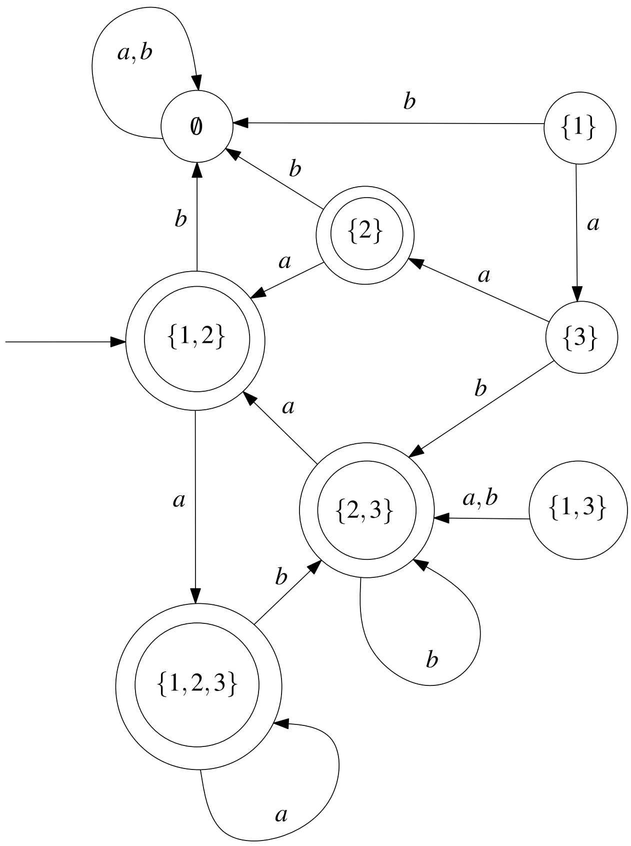
- $\delta' : Q' \times \Sigma \to Q'$ is defined as follows: For each $R \in Q'$ and for each $a \in \Sigma$,

$$\delta'(R, a) = \bigcup_{r \in R} C_\epsilon(\delta(r, a)).$$

In this example δ' is given by

$$\begin{aligned}
\delta'(\emptyset, a) &= \emptyset & \delta'(\emptyset, b) &= \emptyset \\
\delta'(\{1\}, a) &= \{3\} & \delta'(\{1\}, b) &= \emptyset \\
\delta'(\{2\}, a) &= \{1,2\} & \delta'(\{2\}, b) &= \emptyset \\
\delta'(\{3\}, a) &= \{2\} & \delta'(\{3\}, b) &= \{2,3\} \\
\delta'(\{1,2\}, a) &= \{1,2,3\} & \delta'(\{1,2\}, b) &= \emptyset \\
\delta'(\{1,3\}, a) &= \{2,3\} & \delta'(\{1,3\}, b) &= \{2,3\} \\
\delta'(\{2,3\}, a) &= \{1,2\} & \delta'(\{2,3\}, b) &= \{2,3\} \\
\delta'(\{1,2,3\}, a) &= \{1,2,3\} & \delta'(\{1,2,3\}, b) &= \{2,3\}
\end{aligned}$$

The state diagram of the DFA M is as follows:



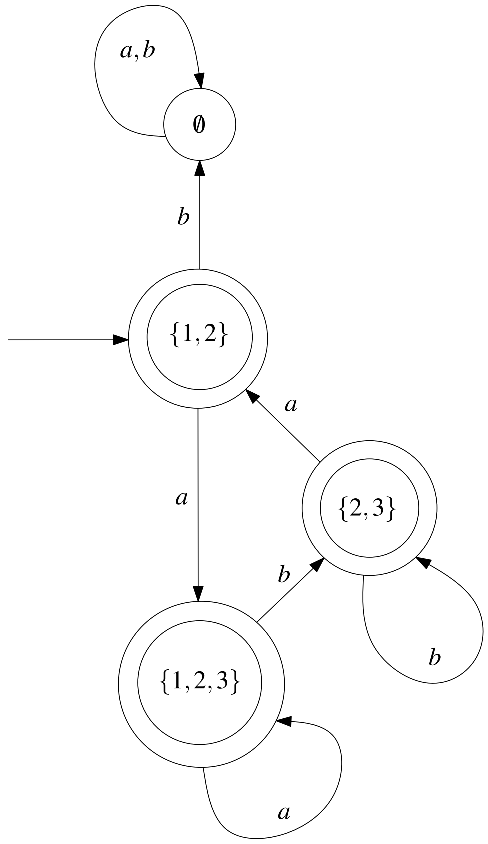
We make the following observations:

- The states $\{1\}$ and $\{1,3\}$ do not have incoming edges. Therefore, these two states cannot be reached from the start state $\{1,2\}$.

- The state $\{3\}$ has only one incoming edge; it comes from the state $\{1\}$. Since $\{1\}$ cannot be reached from the start state, $\{3\}$ cannot be reached from the start state.

- The state $\{2\}$ has only one incoming edge; it comes from the state $\{3\}$. Since $\{3\}$ cannot be reached from the start state, $\{2\}$ cannot be reached from the start state.

Hence, we can remove the four states $\{1\}$, $\{2\}$, $\{3\}$, and $\{1,3\}$. The resulting DFA accepts the same language as the DFA above. This leads to the following state diagram, which depicts a DFA that accepts the same language as the NFA N:



2.6 Closure under the regular operations

In Section 2.3, we have defined the regular operations union, concatenation, and star. We proved in Theorem 2.3.1 that the union of two regular languages is a regular language. We also explained why it is not clear that the concatenation of two regular languages is regular, and that the star of a regular language is regular. In this section, we will see that the concept of NFA, together with Theorem 2.5.2, can be used to give a simple proof of the fact that the regular languages are indeed closed under the regular operations. We start by giving an alternative proof of Theorem 2.3.1:

Theorem 2.6.1 *The set of regular languages is closed under the union operation, i.e., if A_1 and A_2 are regular languages over the same alphabet Σ, then $A_1 \cup A_2$ is also a regular language.*

Proof. Since A_1 is regular, there is, by Theorem 2.5.2, an NFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, such that $A_1 = L(M_1)$. Similarly, there is an NFA $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, such that $A_2 = L(M_2)$. We may assume that $Q_1 \cap Q_2 = \emptyset$, because otherwise, we can give new "names" to the states of Q_1 and Q_2. From these two NFAs, we will construct an NFA $M = (Q, \Sigma, \delta, q_0, F)$, such that $L(M) = A_1 \cup A_2$. The construction is illustrated in Figure 2.1. The NFA M is defined as follows:

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$, where q_0 is a new state.

2. q_0 is the start state of M.

3. $F = F_1 \cup F_2$.

4. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is defined as follows: For any $r \in Q$ and for any $a \in \Sigma_\epsilon$,
$$\delta(r, a) = \begin{cases} \delta_1(r, a) & \text{if } r \in Q_1, \\ \delta_2(r, a) & \text{if } r \in Q_2, \\ \{q_1, q_2\} & \text{if } r = q_0 \text{ and } a = \epsilon, \\ \emptyset & \text{if } r = q_0 \text{ and } a \neq \epsilon. \end{cases}$$

∎

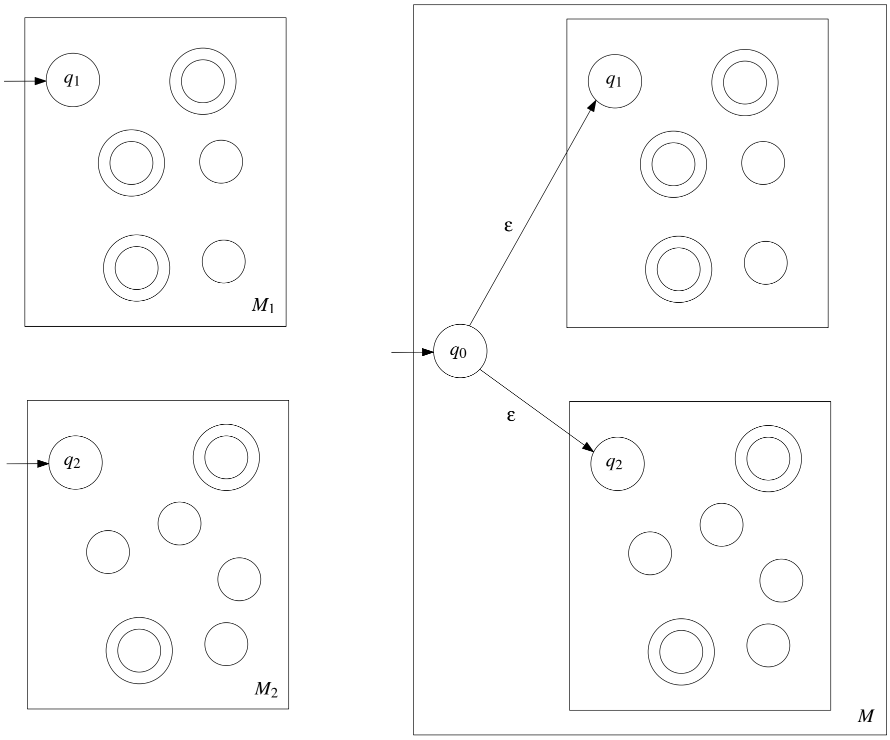


Figure 2.1: *The NFA M accepts $L(M_1) \cup L(M_2)$.*

Theorem 2.6.2 *The set of regular languages is closed under the concatenation operation, i.e., if A_1 and A_2 are regular languages over the same alphabet Σ, then A_1A_2 is also a regular language.*

Proof. Let $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ be an NFA, such that $A_1 = L(M_1)$. Similarly, let $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be an NFA, such that $A_2 = L(M_2)$. As in the proof of Theorem 2.6.1, we may assume that $Q_1 \cap Q_2 = \emptyset$. We will construct an NFA $M = (Q, \Sigma, \delta, q_0, F)$, such that $L(M) = A_1A_2$. The construction is illustrated in Figure 2.2. The NFA M is defined as follows:

1. $Q = Q_1 \cup Q_2$.
2. $q_0 = q_1$.
3. $F = F_2$.

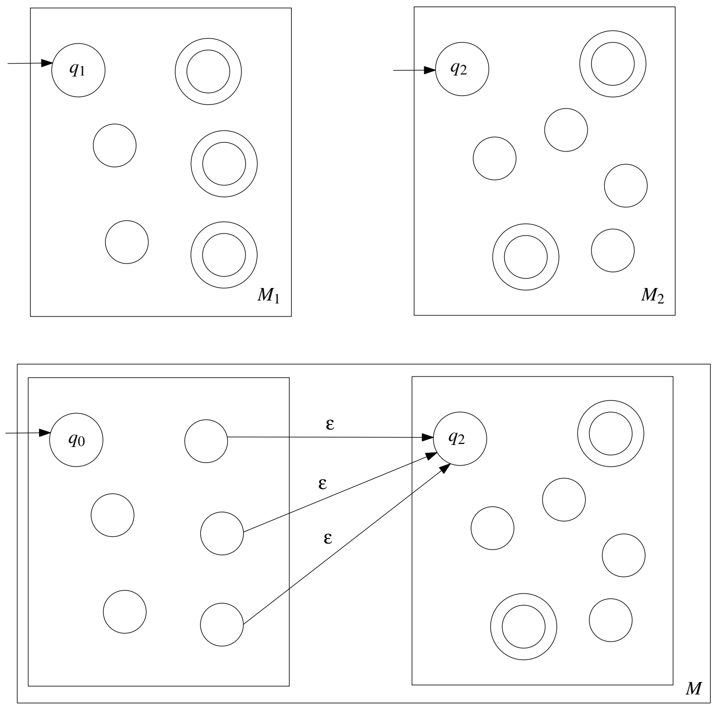


Figure 2.2: *The NFA M accepts $L(M_1)L(M_2)$.*

4. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is defined as follows: For any $r \in Q$ and for any $a \in \Sigma_\epsilon$,

$$\delta(r,a) = \begin{cases} \delta_1(r,a) & \text{if } r \in Q_1 \text{ and } r \notin F_1, \\ \delta_1(r,a) & \text{if } r \in F_1 \text{ and } a \neq \epsilon, \\ \delta_1(r,a) \cup \{q_2\} & \text{if } r \in F_1 \text{ and } a = \epsilon, \\ \delta_2(r,a) & \text{if } r \in Q_2. \end{cases}$$

∎

Theorem 2.6.3 *The set of regular languages is closed under the star operation, i.e., if A is a regular language, then A^* is also a regular language.*

Proof. Let Σ be the alphabet of A and let $N = (Q_1, \Sigma, \delta_1, q_1, F_1)$ be an NFA, such that $A = L(N)$. We will construct an NFA $M = (Q, \Sigma, \delta, q_0, F)$, such that $L(M) = A^*$. The construction is illustrated in Figure 2.3. The NFA M is defined as follows:

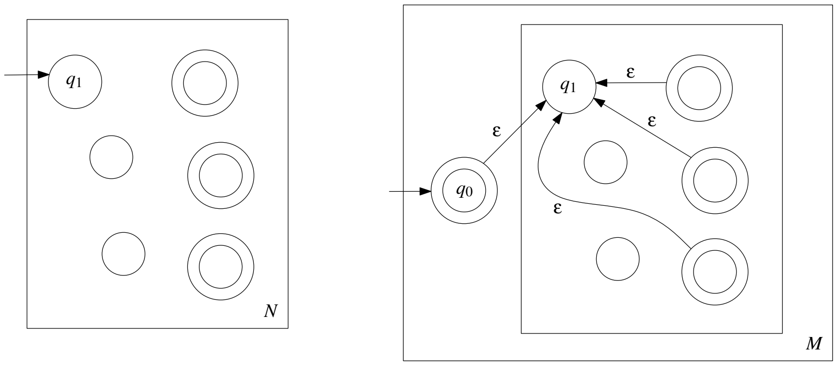


Figure 2.3: *The NFA M accepts $(L(N))^*$.*

1. $Q = \{q_0\} \cup Q_1$, where q_0 is a new state.
2. q_0 is the start state of M.
3. $F = \{q_0\} \cup F_1$. (Since $\epsilon \in A^*$, q_0 has to be an accept state.)
4. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is defined as follows: For any $r \in Q$ and for any $a \in \Sigma_\epsilon$,

$$\delta(r,a) = \begin{cases} \delta_1(r,a) & \text{if } r \in Q_1 \text{ and } r \notin F_1, \\ \delta_1(r,a) & \text{if } r \in F_1 \text{ and } a \neq \epsilon, \\ \delta_1(r,a) \cup \{q_1\} & \text{if } r \in F_1 \text{ and } a = \epsilon, \\ \{q_1\} & \text{if } r = q_0 \text{ and } a = \epsilon, \\ \emptyset & \text{if } r = q_0 \text{ and } a \neq \epsilon. \end{cases}$$

■

In the final theorem of this section, we mention (without proof) two more closure properties of the regular languages:

Theorem 2.6.4 *The set of regular languages is closed under the complement and intersection operations:*

1. *If A is a regular language over the alphabet Σ, then the complement*

$$\overline{A} = \{w \in \Sigma^* : w \notin A\}$$

is also a regular language.

2. *If* A_1 *and* A_2 *are regular languages over the same alphabet* Σ*, then the intersection*

$$A_1 \cap A_2 = \{w \in \Sigma^* : w \in A_1 \text{ and } w \in A_2\}$$

is also a regular language.

2.7 Regular expressions

In this section, we present regular expressions, which are a means to describe languages. As we will see, the class of languages that can be described by regular expressions coincides with the class of regular languages.

Before formally defining the notion of a regular expression, we give some examples. Consider the expression

$$(0 \cup 1)01^*.$$

The language described by this expression is the set of all binary strings

1. that start with either 0 or 1 (this is indicated by $(0 \cup 1)$),
2. for which the second symbol is 0 (this is indicated by 0), and
3. that end with zero or more 1s (this is indicated by 1^*).

That is, the language described by this expression is

$$\{00, 001, 0011, 00111, \ldots, 10, 101, 1011, 10111, \ldots\}.$$

Here are some more examples (in all cases, the alphabet is $\{0, 1\}$):

- The language $\{w : w \text{ contains exactly two 0s}\}$ is described by the expression
$$1^*01^*01^*.$$
- The language $\{w : w \text{ contains at least two 0s}\}$ is described by the expression
$$(0 \cup 1)^*0(0 \cup 1)^*0(0 \cup 1)^*.$$
- The language $\{w : 1011 \text{ is a substring of } w\}$ is described by the expression
$$(0 \cup 1)^*1011(0 \cup 1)^*.$$

- The language $\{w : \text{ the length of } w \text{ is even}\}$ is described by the expression
$$((0 \cup 1)(0 \cup 1))^* .$$
- The language $\{w : \text{ the length of } w \text{ is odd}\}$ is described by the expression
$$(0 \cup 1)\,((0 \cup 1)(0 \cup 1))^* .$$
- The language $\{1011, 0\}$ is described by the expression
$$1011 \cup 0.$$
- The language $\{w : \text{ the first and last symbols of } w \text{ are equal}\}$ is described by the expression
$$0(0 \cup 1)^*0 \cup 1(0 \cup 1)^*1 \cup 0 \cup 1.$$

After these examples, we give a formal (and inductive) definition of *regular expressions*:

Definition 2.7.1 Let Σ be a non-empty alphabet.

1. ϵ is a regular expression.
2. $\emptyset$ is a regular expression.
3. For each $a \in \Sigma$, a is a regular expression.
4. If R_1 and R_2 are regular expressions, then $R_1 \cup R_2$ is a regular expression.
5. If R_1 and R_2 are regular expressions, then $R_1 R_2$ is a regular expression.
6. If R is a regular expression, then R^* is a regular expression.

You can regard 1., 2., and 3. as being the "building blocks" of regular expressions. Items 4., 5., and 6. give rules that can be used to combine regular expressions into new (and "larger") regular expressions. To give an example, we claim that
$$(0 \cup 1)^*101(0 \cup 1)^*$$
is a regular expression (where the alphabet Σ is equal to $\{0, 1\}$). In order to prove this, we have to show that this expression can be "built" using the "rules" given in Definition 2.7.1. Here we go:

- By 3., 0 is a regular expression.
- By 3., 1 is a regular expression.
- Since 0 and 1 are regular expressions, by 4., $0 \cup 1$ is a regular expression.
- Since $0 \cup 1$ is a regular expression, by 6., $(0 \cup 1)^*$ is a regular expression.
- Since 1 and 0 are regular expressions, by 5., 10 is a regular expression.
- Since 10 and 1 are regular expressions, by 5., 101 is a regular expression.
- Since $(0 \cup 1)^*$ and 101 are regular expressions, by 5., $(0 \cup 1)^*101$ is a regular expression.
- Since $(0 \cup 1)^*101$ and $(0 \cup 1)^*$ are regular expressions, by 5., $(0 \cup 1)^*101(0 \cup 1)^*$ is a regular expression.

Next we define the language that is *described* by a regular expression:

Definition 2.7.2 Let Σ be a non-empty alphabet.

1. The regular expression ϵ describes the language $\{\epsilon\}$.
2. The regular expression $\emptyset$ describes the language $\emptyset$.
3. For each $a \in \Sigma$, the regular expression a describes the language $\{a\}$.
4. Let R_1 and R_2 be regular expressions and let L_1 and L_2 be the languages described by them, respectively. The regular expression $R_1 \cup R_2$ describes the language $L_1 \cup L_2$.
5. Let R_1 and R_2 be regular expressions and let L_1 and L_2 be the languages described by them, respectively. The regular expression $R_1 R_2$ describes the language $L_1 L_2$.
6. Let R be a regular expression and let L be the language described by it. The regular expression R^* describes the language L^*.

We consider some examples:

- The regular expression $(0 \cup \epsilon)(1 \cup \epsilon)$ describes the language $\{01, 0, 1, \epsilon\}$.

- The regular expression $0 \cup \epsilon$ describes the language $\{0, \epsilon\}$, whereas the regular expression 1^* describes the language $\{\epsilon, 1, 11, 111, \ldots\}$. Therefore, the regular expression $(0 \cup \epsilon)1^*$ describes the language

$$\{0, 01, 011, 0111, \ldots, \epsilon, 1, 11, 111, \ldots\}.$$

  Observe that this language is also described by the regular expression $01^* \cup 1^*$.

- The regular expression $1^*\emptyset$ describes the empty language, i.e., the language $\emptyset$. (You should convince yourself that this is correct.)

- The regular expression $\emptyset^*$ describes the language $\{\epsilon\}$.

Definition 2.7.3 Let R_1 and R_2 be regular expressions and let L_1 and L_2 be the languages described by them, respectively. If $L_1 = L_2$ (i.e., R_1 and R_2 describe the same language), then we will write $R_1 = R_2$.

Hence, even though $(0 \cup \epsilon)1^*$ and $01^* \cup 1^*$ are different regular expressions, we write

$$(0 \cup \epsilon)1^* = 01^* \cup 1^*,$$

because they describe the same language.

In Section 2.8.2, we will show that every regular language can be described by a regular expression. The proof of this fact is purely algebraic and uses the following algebraic identities involving regular expressions.

Theorem 2.7.4 *Let R_1, R_2, and R_3 be regular expressions. The following identities hold:*

1. $R_1\emptyset = \emptyset R_1 = \emptyset$.
2. $R_1\epsilon = \epsilon R_1 = R_1$.
3. $R_1 \cup \emptyset = \emptyset \cup R_1 = R_1$.
4. $R_1 \cup R_1 = R_1$.
5. $R_1 \cup R_2 = R_2 \cup R_1$.
6. $R_1(R_2 \cup R_3) = R_1R_2 \cup R_1R_3$.

7. $(R_1 \cup R_2)R_3 = R_1R_3 \cup R_2R_3$.
8. $R_1(R_2R_3) = (R_1R_2)R_3$.
9. $\emptyset^* = \epsilon$.
10. $\epsilon^* = \epsilon$.
11. $(\epsilon \cup R_1)^* = R_1^*$.
12. $(\epsilon \cup R_1)(\epsilon \cup R_1)^* = R_1^*$.
13. $R_1^*(\epsilon \cup R_1) = (\epsilon \cup R_1)R_1^* = R_1^*$.
14. $R_1^*R_2 \cup R_2 = R_1^*R_2$.
15. $R_1(R_2R_1)^* = (R_1R_2)^*R_1$.
16. $(R_1 \cup R_2)^* = (R_1^*R_2)^*R_1^* = (R_2^*R_1)^*R_2^*$.

We will not present the (boring) proofs of these identities, but urge you to convince yourself informally that they make perfect sense. To give an example, we mentioned above that

$$(0 \cup \epsilon)1^* = 01^* \cup 1^*.$$

We can verify this identity in the following way:

$$\begin{aligned} (0 \cup \epsilon)1^* &= 01^* \cup \epsilon 1^* && \text{(by identity 7)} \\ &= 01^* \cup 1^* && \text{(by identity 2)} \end{aligned}$$

2.8 Equivalence of regular expressions and regular languages

In the beginning of Section 2.7, we mentioned the following result:

Theorem 2.8.1 *Let L be a language. Then L is regular if and only if there exists a regular expression that describes L.*

The proof of this theorem consists of two parts:

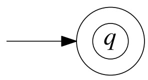
- In Section 2.8.1, we will prove that every regular expression describes a regular language.
- In Section 2.8.2, we will prove that every DFA M can be converted to a regular expression that describes the language $L(M)$.

These two results will prove Theorem 2.8.1.

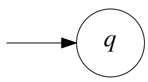
2.8.1 Every regular expression describes a regular language

Let R be an arbitrary regular expression over the alphabet Σ. We will prove that the language described by R is a regular language. The proof is by induction on the structure of R (i.e., by induction on the way R is "built" using the "rules" given in Definition 2.7.1).

The first base case: Assume that $R = \epsilon$. Then R describes the language $\{\epsilon\}$. In order to prove that this language is regular, it suffices, by Theorem 2.5.2, to construct an NFA $M = (Q, \Sigma, \delta, q, F)$ that accepts this language. This NFA is obtained by defining $Q = \{q\}$, q is the start state, $F = \{q\}$, and $\delta(q, a) = \emptyset$ for all $a \in \Sigma_\epsilon$. The figure below gives the state diagram of M:



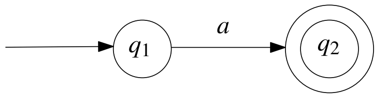
The second base case: Assume that $R = \emptyset$. Then R describes the language $\emptyset$. In order to prove that this language is regular, it suffices, by Theorem 2.5.2, to construct an NFA $M = (Q, \Sigma, \delta, q, F)$ that accepts this language. This NFA is obtained by defining $Q = \{q\}$, q is the start state, $F = \emptyset$, and $\delta(q, a) = \emptyset$ for all $a \in \Sigma_\epsilon$. The figure below gives the state diagram of M:



The third base case: Let $a \in \Sigma$ and assume that $R = a$. Then R describes the language $\{a\}$. In order to prove that this language is regular, it suffices, by Theorem 2.5.2, to construct an NFA $M = (Q, \Sigma, \delta, q_1, F)$ that accepts this language. This NFA is obtained by defining $Q = \{q_1, q_2\}$, q_1 is the start state, $F = \{q_2\}$, and

$$\begin{aligned}
\delta(q_1, a) &= \{q_2\}, \\
\delta(q_1, b) &= \emptyset \text{ for all } b \in \Sigma_\epsilon \setminus \{a\}, \\
\delta(q_2, b) &= \emptyset \text{ for all } b \in \Sigma_\epsilon.
\end{aligned}$$

The figure below gives the state diagram of M:



The first case of the induction step: Assume that $R = R_1 \cup R_2$, where R_1 and R_2 are regular expressions. Let L_1 and L_2 be the languages described by R_1 and R_2, respectively, and assume that L_1 and L_2 are regular. Then R describes the language $L_1 \cup L_2$, which, by Theorem 2.6.1, is regular.

The second case of the induction step: Assume that $R = R_1R_2$, where R_1 and R_2 are regular expressions. Let L_1 and L_2 be the languages described by R_1 and R_2, respectively, and assume that L_1 and L_2 are regular. Then R describes the language L_1L_2, which, by Theorem 2.6.2, is regular.

The third case of the induction step: Assume that $R = (R_1)^*$, where R_1 is a regular expression. Let L_1 be the language described by R_1 and assume that L_1 is regular. Then R describes the language $(L_1)^*$, which, by Theorem 2.6.3, is regular.

This concludes the proof of the claim that every regular expression describes a regular language.

To give an example, consider the regular expression

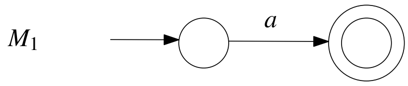
$$(ab \cup a)^*,$$

where the alphabet is $\{a, b\}$. We will prove that this regular expression describes a regular language, by constructing an NFA that accepts the language described by this regular expression. Observe how the regular expression is "built":

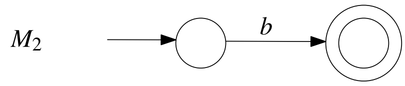
- Take the regular expressions a and b, and combine them into the regular expression ab.

- Take the regular expressions ab and a, and combine them into the regular expression $ab \cup a$.
- Take the regular expression $ab \cup a$, and transform it into the regular expression $(ab \cup a)^*$.

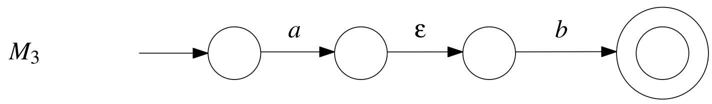
First, we construct an NFA M_1 that accepts the language described by the regular expression a:



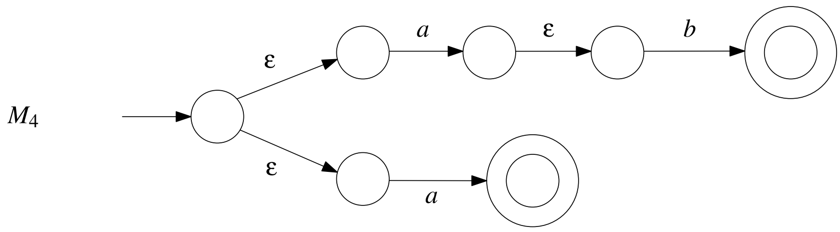
Next, we construct an NFA M_2 that accepts the language described by the regular expression b:



Next, we apply the construction given in the proof of Theorem 2.6.2 to M_1 and M_2. This gives an NFA M_3 that accepts the language described by the regular expression ab:

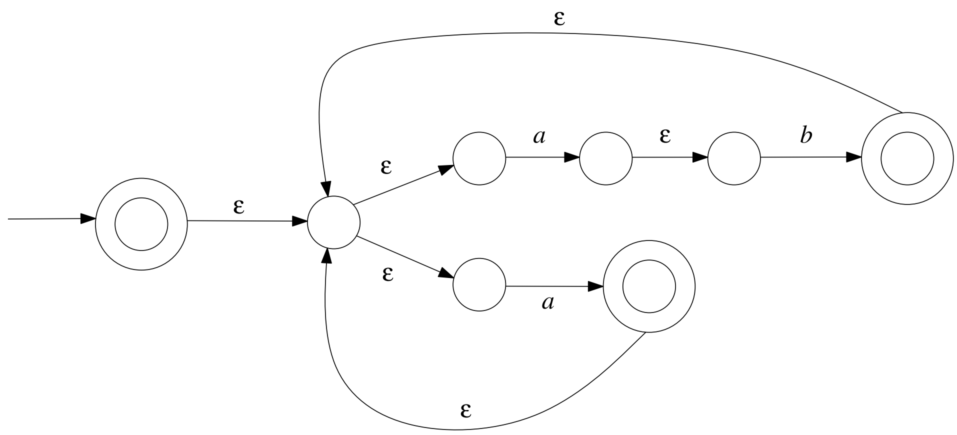


Next, we apply the construction given in the proof of Theorem 2.6.1 to M_3 and M_1. This gives an NFA M_4 that accepts the language described by the regular expression $ab \cup a$:



Finally, we apply the construction given in the proof of Theorem 2.6.3 to M_4. This gives an NFA M_5 that accepts the language described by the regular expression $(ab \cup a)^*$:

M_5



2.8.2 Converting a DFA to a regular expression

In this section, we will prove that every DFA M can be converted to a regular expression that describes the language $L(M)$. In order to prove this result, we need to solve recurrence relations involving languages.

Solving recurrence relations

Let Σ be an alphabet, let B and C be "known" languages in Σ^* such that $\epsilon \notin B$, and let L be an "unknown" language such that

$$L = BL \cup C.$$

Can we "solve" this equation for L? That is, can we express L in terms of B and C?

Consider an arbitrary string u in L. We are going to determine how u looks like. Since $u \in L$ and $L = BL \cup C$, we know that u is a string in $BL \cup C$. Hence, there are two possibilities for u.

1. u is an element of C.

2. u is an element of BL. In this case, there are strings $b \in B$ and $v \in L$ such that $u = bv$. Since $\epsilon \notin B$, we have $b \neq \epsilon$ and, therefore, $|v| < |u|$. (Recall that $|v|$ denotes the length, i.e., the number of symbols, of the string v.) Since v is a string in L, which is equal to $BL \cup C$, v is a string in $BL \cup C$. Hence, there are two possibilities for v.

(a) v is an element of C. In this case,

$$u = bv, \text{ where } b \in B \text{ and } v \in C\text{; thus, } u \in BC.$$

(b) v is an element of BL. In this case, there are strings $b' \in B$ and $w \in L$ such that $v = b'w$. Since $\epsilon \notin B$, we have $b' \neq \epsilon$ and, therefore, $|w| < |v|$. Since w is a string in L, which is equal to $BL \cup C$, w is a string in $BL \cup C$. Hence, there are two possibilities for w.

i. w is an element of C. In this case,

$$u = bb'w, \text{ where } b, b' \in B \text{ and } w \in C\text{; thus, } u \in BBC.$$

ii. w is an element of BL. In this case, there are strings $b'' \in B$ and $x \in L$ such that $w = b''x$. Since $\epsilon \notin B$, we have $b'' \neq \epsilon$ and, therefore, $|x| < |w|$. Since x is a string in L, which is equal to $BL \cup C$, x is a string in $BL \cup C$. Hence, there are two possibilities for x.

A. x is an element of C. In this case,

$$u = bb'b''x, \text{ where } b, b', b'' \in B \text{ and } x \in C\text{; thus, } u \in BBBC.$$

B. x is an element of BL. Etc., etc.

This process hopefully convinces you that any string u in L can be written as the concatenation of zero or more strings in B, followed by one string in C. In fact, L consists of exactly those strings having this property:

Lemma 2.8.2 *Let Σ be an alphabet, and let B, C, and L be languages in Σ^* such that $\epsilon \notin B$ and*

$$L = BL \cup C.$$

Then

$$L = B^*C.$$

Proof. First, we show that $B^*C \subseteq L$. Let u be an arbitrary string in B^*C. Then u is the concatenation of k strings of B, for some $k \geq 0$, followed by one string of C. We proceed by induction on k.

The base case is when $k = 0$. In this case, u is a string in C. Hence, u is a string in $BL \cup C$. Since $BL \cup C = L$, it follows that u is a string in L.

Now let $k \geq 1$. Then we can write $u = vwc$, where v is a string in B, w is the concatenation of $k-1$ strings of B, and c is a string of C. Define $y = wc$. Observe that y is the concatenation of $k-1$ strings of B followed by one string of C. Therefore, by induction, the string y is an element of L. Hence, $u = vy$, where v is a string in B and y is a string in L. This shows that u is a string in BL. Hence, u is a string in $BL \cup C$. Since $BL \cup C = L$, it follows that u is a string in L. This completes the proof that $B^*C \subseteq L$.

It remains to show that $L \subseteq B^*C$. Let u be an arbitrary string in L, and let ℓ be its length (i.e., ℓ is the number of symbols in u). We prove by induction on ℓ that u is a string in B^*C.

The base case is when $\ell = 0$. Then $u = \epsilon$. Since $u \in L$ and $L = BL \cup C$, u is a string in $BL \cup C$. Since $\epsilon \notin B$, u cannot be a string in BL. Hence, u must be a string in C. Since $C \subseteq B^*C$, it follows that u is a string in B^*C.

Let $\ell \geq 1$. If u is a string in C, then u is a string in B^*C and we are done. So assume that u is not a string in C. Since $u \in L$ and $L = BL \cup C$, u is a string in BL. Hence, there are strings $b \in B$ and $v \in L$ such that $u = bv$. Since $\epsilon \notin B$, the length of b is at least one; hence, the length of v is less than the length of u. By induction, v is a string in B^*C. Hence, $u = bv$, where $b \in B$ and $v \in B^*C$. This shows that $u \in B(B^*C)$. Since $B(B^*C) \subseteq B^*C$, it follows that $u \in B^*C$. ∎

Note that Lemma 2.8.2 holds for *any* language B that does not contain the empty string ϵ. As an example, assume that $B = \emptyset$. Then the language L satisfies the equation

$$L = BL \cup C = \emptyset L \cup C.$$

Using Theorem 2.7.4, this equation becomes

$$L = \emptyset \cup C = C.$$

We now show that Lemma 2.8.2 also implies that $L = C$: Since $\epsilon \notin B$, Lemma 2.8.2 implies that $L = B^*C$, which, using Theorem 2.7.4, becomes

$$L = B^*C = \emptyset^*C = \epsilon C = C.$$

The conversion

We will now use Lemma 2.8.2 to prove that every DFA can be converted to a regular expression.

Let $M = (Q, \Sigma, \delta, q, F)$ be an arbitrary deterministic finite automaton. We will show that there exists a regular expression that describes the language $L(M)$.

For each state $r \in Q$, we define

$$L_r = \{w \in \Sigma^* : \begin{array}{l}\text{the path in the state diagram of } M \text{ that starts} \\ \text{in state } r \text{ and that corresponds to } w \text{ ends in a} \\ \text{state of } F\ \}.\end{array}$$

In words, L_r is the language accepted by M, *if r were the start state.*

We will show that each such language L_r can be described by a regular expression. Since $L(M) = L_q$, this will prove that $L(M)$ can be described by a regular expression.

The basic idea is to set up equations for the languages L_r, which we then solve using Lemma 2.8.2. We claim that

$$L_r = \bigcup_{a \in \Sigma} a \cdot L_{\delta(r,a)} \qquad \text{if } r \not\in F. \tag{2.2}$$

Why is this true? Let w be a string in L_r. Then the path P in the state diagram of M that starts in state r and that corresponds to w ends in a state of F. Since $r \not\in F$, this path contains at least one edge. Let r' be the state that follows the first state (i.e., r) of P. Then $r' = \delta(r, b)$ for some symbol $b \in \Sigma$. Hence, b is the first symbol of w. Write $w = bv$, where v is the remaining part of w. Then the path $P' = P \setminus \{r\}$ in the state diagram of M that starts in state r' and that corresponds to v ends in a state of F. Therefore, $v \in L_{r'} = L_{\delta(r,b)}$. Hence,

$$w \in b \cdot L_{\delta(r,b)} \subseteq \bigcup_{a \in \Sigma} a \cdot L_{\delta(r,a)}.$$

Conversely, let w be a string in $\bigcup_{a \in \Sigma} a \cdot L_{\delta(r,a)}$. Then there is a symbol $b \in \Sigma$ and a string $v \in L_{\delta(r,b)}$ such that $w = bv$. Let P' be the path in the state diagram of M that starts in state $\delta(r, b)$ and that corresponds to v. Since $v \in L_{\delta(r,b)}$, this path ends in a state of F. Let P be the path in the state diagram of M that starts in r, follows the edge to $\delta(r, b)$, and then follows P'. This path P corresponds to w and ends in a state of F. Therefore, $w \in L_r$. This proves the correctness of (2.2).

Similarly, we can prove that

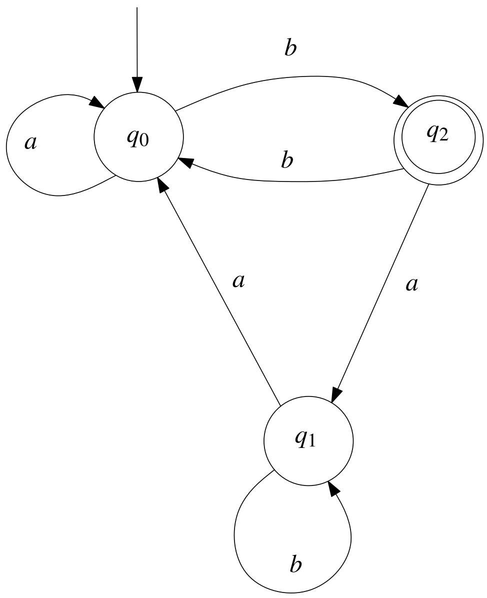
$$L_r = \epsilon \cup \left(\bigcup_{a \in \Sigma} a \cdot L_{\delta(r,a)} \right) \qquad \text{if } r \in F. \tag{2.3}$$

So we now have a set of equations in the "unknowns" L_r, for $r \in Q$. The number of equations is equal to the size of Q. In other words, the number of equations is equal to the number of unknowns. The regular expression for $L(M) = L_q$ is obtained by solving these equations using Lemma 2.8.2.

Of course, we have to convince ourselves that these equations have a solution for any given DFA. Before we deal with this issue, we give an example.

An example

Consider the deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, q_0 is the start state, $F = \{q_2\}$, and δ is given in the state diagram below. We show how to obtain the regular expression that describes the language accepted by M.



For this case, (2.2) and (2.3) give the following equations:

$$\begin{cases} L_{q_0} &= a \cdot L_{q_0} \cup b \cdot L_{q_2} \\ L_{q_1} &= a \cdot L_{q_0} \cup b \cdot L_{q_1} \\ L_{q_2} &= \epsilon \cup a \cdot L_{q_1} \cup b \cdot L_{q_0} \end{cases}$$

In the third equation, L_{q_2} is expressed in terms of L_{q_0} and L_{q_1}. Hence, if we substitute the third equation into the first one, and use Theorem 2.7.4, then we get

$$\begin{aligned} L_{q_0} &= a \cdot L_{q_0} \cup b \cdot (\epsilon \cup a \cdot L_{q_1} \cup b \cdot L_{q_0}) \\ &= (a \cup bb) \cdot L_{q_0} \cup ba \cdot L_{q_1} \cup b. \end{aligned}$$

We obtain the following set of equations.

$$\begin{cases} L_{q_0} = (a \cup bb) \cdot L_{q_0} \cup ba \cdot L_{q_1} \cup b \\ L_{q_1} = b \cdot L_{q_1} \cup a \cdot L_{q_0} \end{cases}$$

Let $L = L_{q_1}$, $B = b$, and $C = a \cdot L_{q_0}$. Then $\epsilon \notin B$ and the second equation reads $L = BL \cup C$. Hence, by Lemma 2.8.2,

$$L_{q_1} = L = B^*C = b^*a \cdot L_{q_0}.$$

If we substitute L_{q_1} into the first equation, then we get (again using Theorem 2.7.4)

$$\begin{aligned} L_{q_0} &= (a \cup bb) \cdot L_{q_0} \cup ba \cdot b^*a \cdot L_{q_0} \cup b \\ &= (a \cup bb \cup bab^*a)L_{q_0} \cup b. \end{aligned}$$

Again applying Lemma 2.8.2, this time with $L = L_{q_0}$, $B = a \cup bb \cup bab^*a$ and $C = b$, gives

$$L_{q_0} = (a \cup bb \cup bab^*a)^* \, b.$$

Thus, the regular expression that describes the language accepted by M is

$$(a \cup bb \cup bab^*a)^* \, b.$$

Completing the correctness of the conversion

It remains to prove that, for any DFA, the system of equations (2.2) and (2.3) can be solved. This will follow from the following (more general) lemma. (You should verify that the equations (2.2) and (2.3) are in the form as specified in this lemma.)

Lemma 2.8.3 *Let $n \geq 1$ be an integer and, for $1 \leq i \leq n$ and $1 \leq j \leq n$, let B_{ij} and C_i be regular expressions such that $\epsilon \notin B_{ij}$. Let $L_1, L_2, \ldots, L_n$ be languages that satisfy*

$$L_i = \left(\bigcup_{j=1}^{n} B_{ij} L_j \right) \cup C_i \text{ for } 1 \leq i \leq n.$$

Then L_1 can be expressed as a regular expression only involving the regular expressions B_{ij} and C_i.

Proof. The proof is by induction on n. The base case is when $n = 1$. In this case, we have

$$L_1 = B_{11} L_1 \cup C_1.$$

Since $\epsilon \notin B_{11}$, it follows from Lemma 2.8.2 that $L_1 = B_{11}^* C_1$. This proves the base case.

Let $n \geq 2$ and assume the lemma is true for $n - 1$. We have

$$\begin{aligned} L_n &= \left(\bigcup_{j=1}^{n} B_{nj} L_j \right) \cup C_n \\ &= B_{nn} L_n \cup \left(\bigcup_{j=1}^{n-1} B_{nj} L_j \right) \cup C_n. \end{aligned}$$

Since $\epsilon \notin B_{nn}$, it follows from Lemma 2.8.2 that

$$\begin{aligned} L_n &= B_{nn}^* \left(\left(\bigcup_{j=1}^{n-1} B_{nj} L_j \right) \cup C_n \right) \\ &= B_{nn}^* \left(\bigcup_{j=1}^{n-1} B_{nj} L_j \right) \cup B_{nn}^* C_n \\ &= \left(\bigcup_{j=1}^{n-1} B_{nn}^* B_{nj} L_j \right) \cup B_{nn}^* C_n \end{aligned}$$

By substituting this equation for L_n into the equations for L_i, $1 \leq i \leq n-1$, we obtain

$$\begin{aligned} L_i &= \left(\bigcup_{j=1}^{n} B_{ij} L_j \right) \cup C_i \\ &= B_{in} L_n \cup \left(\bigcup_{j=1}^{n-1} B_{ij} L_j \right) \cup C_i \\ &= \left(\bigcup_{j=1}^{n-1} \left(B_{in} B_{nn}^* B_{nj} \cup B_{ij} \right) L_j \right) \cup B_{in} B_{nn}^* C_n \cup C_i . \end{aligned}$$

Thus, we have obtained $n-1$ equations in $L_1, L_2, \ldots, L_{n-1}$. Since $\epsilon \notin B_{in} B_{nn}^* B_{nj} \cup B_{ij}$, it follows from the induction hypothesis that L_1 can be expressed as a regular expression only involving the regular expressions B_{ij} and C_i. ∎

2.9 The pumping lemma and nonregular languages

In the previous sections, we have seen that the class of regular languages is closed under various operations, and that these languages can be described by (deterministic or nondeterministic) finite automata and regular expressions. These properties helped in developing techniques for showing that a language is regular. In this section, we will present a tool that can be used to prove that certain languages are *not* regular. Observe that for a regular language,

1. the amount of memory that is needed to determine whether or not a given string is in the language is finite and independent of the length of the string, and

2. if the language consists of an infinite number of strings, then this language should contain infinite subsets having a fairly repetitive structure.

Intuitively, languages that do not follow 1. or 2. should be nonregular. For example, consider the language

$$\{0^n 1^n : n \geq 0\}.$$

This language should be nonregular, because it seems unlikely that a DFA can remember how many 0s it has seen when it has reached the border between the 0s and the 1s. Similarly the language

$$\{0^n : \ n \text{ is a prime number}\}$$

should be nonregular, because the prime numbers do not seem to have any repetitive structure that can be used by a DFA. To be more rigorous about this, we will establish a property that all regular languages must possess. This property is called the *pumping lemma*. If a language does not have this property, then it must be nonregular.

The pumping lemma states that any sufficiently long string in a regular language can be *pumped*, i.e., there is a section in that string that can be repeated any number of times, so that the resulting strings are all in the language.

Theorem 2.9.1 (Pumping Lemma for Regular Languages) *Let A be a regular language. Then there exists an integer $p \geq 1$, called the* pumping length*, such that the following holds: Every string s in A, with $|s| \geq p$, can be written as $s = xyz$, such that*

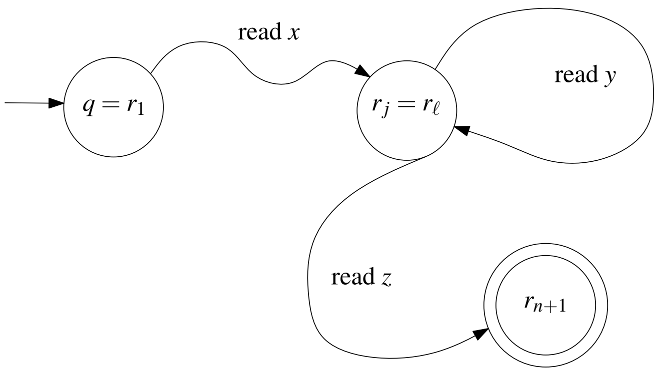
1. *$y \neq \epsilon$ (i.e., $|y| \geq 1$),*
2. *$|xy| \leq p$, and*
3. *for all $i \geq 0$, $xy^iz \in A$.*

In words, the pumping lemma states that by replacing the portion y in s by zero or more copies of it, the resulting string is still in the language A.

Proof. Let Σ be the alphabet of A. Since A is a regular language, there exists a DFA $M = (Q, \Sigma, \delta, q, F)$ that accepts A. We define p to be the number of states in Q.

Let $s = s_1 s_2 \dots s_n$ be an arbitrary string in A such that $n \geq p$. Define $r_1 = q$, $r_2 = \delta(r_1, s_1)$, $r_3 = \delta(r_2, s_2)$, ..., $r_{n+1} = \delta(r_n, s_n)$. Thus, when the DFA M reads the string s from left to right, it visits the states $r_1, r_2, \dots, r_{n+1}$. Since s is a string in A, we know that r_{n+1} belongs to F.

Consider the first $p+1$ states $r_1, r_2, \dots, r_{p+1}$ in this sequence. Since the number of states of M is equal to p, the pigeonhole principle implies that there must be a state that occurs twice in this sequence. That is, there are indices j and ℓ such that $1 \leq j < \ell \leq p+1$ and $r_j = r_\ell$.



We define $x = s_1 s_2 \dots s_{j-1}$, $y = s_j \dots s_{\ell-1}$, and $z = s_\ell \dots s_n$. Since $j < \ell$, we have $y \neq \epsilon$, proving the first claim in the theorem. Since $\ell \leq p+1$, we have $|xy| = \ell - 1 \leq p$, proving the second claim in the theorem. To see that the third claim also holds, recall that the string $s = xyz$ is accepted by M. While reading x, M moves from the start state q to state r_j. While reading y, it moves from state r_j to state $r_\ell = r_j$, i.e., after having read y, M is again in state r_j. While reading z, M moves from state r_j to the accept state r_{n+1}. Therefore, the substring y can be repeated any number $i \geq 0$ of times, and the corresponding string xy^iz will still be accepted by M. It follows that $xy^iz \in A$ for all $i \geq 0$. ∎

2.9.1 Applications of the pumping lemma

First example

Consider the language

$$A = \{0^n 1^n : n \geq 0\}.$$

We will prove by contradiction that A is not a regular language.

Assume that A is a regular language. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. Consider the string $s = 0^p 1^p$. It is clear that $s \in A$ and $|s| = 2p \geq p$. Hence, by the pumping lemma, s can be written as $s = xyz$, where $y \neq \epsilon$, $|xy| \leq p$, and $xy^iz \in A$ for all $i \geq 0$.

Observe that, since $|xy| \leq p$, the string y contains only 0s. Moreover, since $y \neq \epsilon$, y contains at least one 0. But now we are in trouble: None of the strings $xy^0z = xz$, $xy^2z = xyyz$, $xy^3z = xyyyz$, $\dots$, is contained in A. However, by the pumping lemma, all these strings must be in A. Hence, we have a contradiction and we conclude that A is not a regular language.

Second example

Consider the language

$$A = \{w \in \{0,1\}^* : \text{ the number of 0s in } w \text{ equals the number of 1s in } w\}.$$

Again, we prove by contradiction that A is not a regular language.

Assume that A is a regular language. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. Consider the string $s = 0^p1^p$. Then $s \in A$ and $|s| = 2p \geq p$. By the pumping lemma, s can be written as $s = xyz$, where $y \neq \epsilon$, $|xy| \leq p$, and $xy^iz \in A$ for all $i \geq 0$.

Since $|xy| \leq p$, the string y contains only 0s. Since $y \neq \epsilon$, y contains at least one 0. Therefore, the string $xy^2z = xyyz$ contains more 0s than 1s, which implies that this string is not contained in A. But, by the pumping lemma, this string is contained in A. This is a contradiction and, therefore, A is not a regular language.

Third example

Consider the language

$$A = \{ww : w \in \{0,1\}^*\}.$$

We prove by contradiction that A is not a regular language.

Assume that A is a regular language. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. Consider the string $s = 0^p10^p1$. Then $s \in A$ and $|s| = 2p + 2 \geq p$. By the pumping lemma, s can be written as $s = xyz$, where $y \neq \epsilon$, $|xy| \leq p$, and $xy^iz \in A$ for all $i \geq 0$.

Since $|xy| \leq p$, the string y contains only 0s. Since $y \neq \epsilon$, y contains at least one 0. Therefore, the string $xy^2z = xyyz$ is not contained in A. But, by the pumping lemma, this string is contained in A. This is a contradiction and, therefore, A is not a regular language.

You should convince yourself that by choosing $s = 0^{2p}$ (which is a string in A whose length is at least p), we do not obtain a contradiction. The reason is that the string y may have an even length. Thus, 0^{2p} is the "wrong" string for showing that A is not regular. By choosing $s = 0^p10^p1$, we do obtain a contradiction; thus, this is the "correct" string for showing that A is not regular.

Fourth example

Consider the language

$$A = \{0^m 1^n : m > n \geq 0\}.$$

We prove by contradiction that A is not a regular language.

Assume that A is a regular language. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. Consider the string $s = 0^{p+1}1^p$. Then $s \in A$ and $|s| = 2p + 1 \geq p$. By the pumping lemma, s can be written as $s = xyz$, where $y \neq \epsilon$, $|xy| \leq p$, and $xy^iz \in A$ for all $i \geq 0$.

Since $|xy| \leq p$, the string y contains only 0s. Since $y \neq \epsilon$, y contains at least one 0. Consider the string $xy^0z = xz$. The number of 1s in this string is equal to p, whereas the number of 0s is at most equal to p. Therefore, the string xy^0z is not contained in A. But, by the pumping lemma, this string is contained in A. This is a contradiction and, therefore, A is not a regular language.

Fifth example

Consider the language

$$A = \{1^{n^2} : n \geq 0\}.$$

We prove by contradiction that A is not a regular language.

Assume that A is a regular language. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. Consider the string $s = 1^{p^2}$. Then $s \in A$ and $|s| = p^2 \geq p$. By the pumping lemma, s can be written as $s = xyz$, where $y \neq \epsilon$, $|xy| \leq p$, and $xy^iz \in A$ for all $i \geq 0$.

Observe that

$$|s| = |xyz| = p^2$$

and

$$|xy^2z| = |xyyz| = |xyz| + |y| = p^2 + |y|.$$

Since $|xy| \leq p$, we have $|y| \leq p$. Since $y \neq \epsilon$, we have $|y| \geq 1$. It follows that

$$p^2 < |xy^2z| \leq p^2 + p < (p+1)^2.$$

Hence, the length of the string xy^2z is strictly between two consecutive squares. It follows that this length is not a square and, therefore, xy^2z is not contained in A. But, by the pumping lemma, this string is contained in A. This is a contradiction and, therefore, A is not a regular language.

Sixth example

Consider the language

$$A = \{1^n : \ n \text{ is a prime number}\}.$$

We prove by contradiction that A is not a regular language.

Assume that A is a regular language. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. Let $n \geq p$ be a prime number, and consider the string $s = 1^n$. Then $s \in A$ and $|s| = n \geq p$. By the pumping lemma, s can be written as $s = xyz$, where $y \neq \epsilon$, $|xy| \leq p$, and $xy^iz \in A$ for all $i \geq 0$.

Let k be the integer such that $y = 1^k$. Since $y \neq \epsilon$, we have $k \geq 1$. For each $i \geq 0$, $n + (i-1)k$ is a prime number, because $xy^iz = 1^{n+(i-1)k} \in A$. For $i = n+1$, however, we have

$$n + (i-1)k = n + nk = n(1+k),$$

which is not a prime number, because $n \geq 2$ and $1 + k \geq 2$. This is a contradiction and, therefore, A is not a regular language.

Seventh example

Consider the language

$$A = \{w \in \{0,1\}^* : \ \begin{array}{l}\text{the number of occurrences of 01 in } w \text{ is equal to} \\ \text{the number of occurrences of 10 in } w \ \}.\end{array}$$

Since this language has the same flavor as the one in the second example, we may suspect that A is not a regular language. This is, however, not true: As we will show, A is a regular language.

The key property is the following one: Let w be an arbitrary string in $\{0,1\}^*$. Then

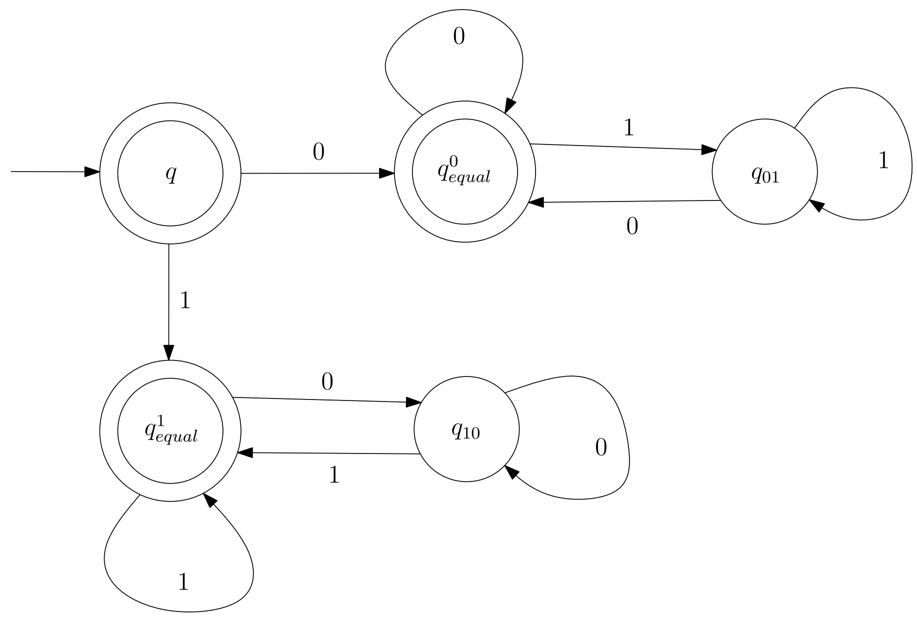
> the absolute value of the number of occurrences of 01 in w minus the number of occurrences of 10 in w is at most one.

This property holds, because between any two consecutive occurrences of 01, there must be exactly one occurrence of 10. Similarly, between any two consecutive occurrences of 10, there must be exactly one occurrence of 01.

We will construct a DFA that accepts A. This DFA uses the following five states:

- q: start state; no symbol has been read.
- q_{01}: the last symbol read was 1; in the part of the string read so far, the number of occurrences of 01 is one more than the number of occurrences of 10.
- q_{10}: the last symbol read was 0; in the part of the string read so far, the number of occurrences of 10 is one more than the number of occurrences of 01.
- q^0_{equal}: the last symbol read was 0; in the part of the string read so far, the number of occurrences of 01 is equal to the number of occurrences of 10.
- q^1_{equal}: the last symbol read was 1; in the part of the string read so far, the number of occurrences of 01 is equal to the number of occurrences of 10.

The set of accept states is equal to $\{q, q^0_{equal}, q^1_{equal}\}$. The state diagram of the DFA is given below.



In fact, the key property mentioned above implies that the language A consists of the empty string ϵ and all non-empty binary strings that start and end with the same symbol. As a result, A is the language described by the regular expression

$$\epsilon \cup 0 \cup 1 \cup 0(0 \cup 1)^*0 \cup 1(0 \cup 1)^*1.$$

This gives an alternative proof for the fact that A is a regular language.

Eighth example

Consider the language

$$L = \{w \in \{0,1\}^* : \ w \text{ is the binary representation of a prime number}\}.$$

We assume that for any positive integer, the leftmost bit in its binary representation is 1. In other words, we assume that there are no 0's added to the left of such a binary representation. Thus,

$$L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}.$$

We will prove that L is not a regular language.

Assume that L is a regular language. Let $p \geq 1$ be the pumping length. Let $N > 2^p$ be a prime number and let $s \in \{0,1\}^*$ be the binary representation of N. Observe that $|s| \geq p+1$. Also, the leftmost and rightmost bits of s are 1.

Since $s \in L$ and $|s| \geq p+1 \geq p$, the Pumping Lemma implies that we can write $s = xyz$, such that

1. $|y| \geq 1$,
2. $|xy| \leq p$ (and, thus, $|z| \geq 1$), and
3. for all $i \geq 0$, $xy^iz \in L$, i.e., xy^iz is the binary representation of a prime number.

Define A, B, and C to be the integers whose binary representations are x, y, and z, respectively. Note that both y and z may have leading 0's. In fact, y may be a string consisting of 0's only, in which case $B = 0$. However, since the rightmost bit of z is 1, we have $C \geq 1$. Observe that

$$N = C + B \cdot 2^{|z|} + A \cdot 2^{|z|+|y|}. \tag{2.4}$$

Let $i = N$, consider the bitstring $xy^iz = xy^Nz$, and let M be the prime number whose binary representation is given by this bitstring. Then,

$$\begin{aligned} M &= C + \sum_{k=0}^{N-1} B \cdot 2^{|z|+k|y|} + A \cdot 2^{|z|+N|y|} \\ &= C + B \cdot 2^{|z|} \sum_{k=0}^{N-1} 2^{k|y|} + A \cdot 2^{|z|+N|y|}. \end{aligned}$$

Let

$$T = \sum_{k=0}^{N-1} 2^{k|y|}.$$

Then

$$\left(2^{|y|} - 1\right) T = 2^{N|y|} - 1. \tag{2.5}$$

By Fermat's Little Theorem, we have

$$2^N \equiv 2 \pmod{N},$$

implying that

$$2^{N|y|} - 1 = \left(2^N\right)^{|y|} - 1 \equiv 2^{|y|} - 1 \pmod{N}.$$

Thus, (2.5) implies that

$$\left(2^{|y|} - 1\right) T \equiv 2^{|y|} - 1 \pmod{N}. \tag{2.6}$$

Observe that $2^{|y|} \leq 2^p < N$, because $|y| \leq |xy| \leq p$. Also, $2^{|y|} \geq 2$, because $y \neq \epsilon$. It follows that

$$1 \leq 2^{|y|} - 1 < N,$$

implying that

$$2^{|y|} - 1 \not\equiv 0 \pmod{N}.$$

This, together with (2.6), implies that

$$T \equiv 1 \pmod{N}.$$

Since

$$M = C + B \cdot 2^{|z|} \cdot T + A \cdot 2^{|z|+N|y|},$$

it follows that

$$M \equiv C + B \cdot 2^{|z|} + A \cdot 2^{|z|+|y|} \pmod{N}.$$

This, together with (2.4), implies that

$$M \equiv 0 \pmod{N},$$

i.e., N divides M. Since $M > N$, we conclude that M is not a prime number, which is a contradiction. Thus, the language L is not regular.

2.10 Higman's Theorem

Let Σ be a finite alphabet. For any two strings x and y in Σ^*, we say that x is a *subsequence* of y, if x can be obtained by deleting zero or more symbols from y. For example, 10110 is a subsequence of 0010010101010001. For any language $L \subseteq \Sigma^*$, we define

$$SUBSEQ(L) := \{x : \text{ there exists a } y \in L \text{ such that } x \text{ is a subsequence of } y\}.$$

That is, $SUBSEQ(L)$ is the language consisting of the subsequences of all strings in L. In 1952, Higman proved the following result:

Theorem 2.10.1 (Higman) *For any finite alphabet Σ and for any language $L \subseteq \Sigma^*$, the language $SUBSEQ(L)$ is regular.*

2.10.1 Dickson's Theorem

Our proof of Higman's Theorem will use a theorem that was proved in 1913 by Dickson.

Recall that $\mathbb{N}$ denotes the set of positive integers. Let $n \in \mathbb{N}$. For any two points $p = (p_1, p_2, \ldots, p_n)$ and $q = (q_1, q_2, \ldots, q_n)$ in $\mathbb{N}^n$, we say that p is *dominated* by q, if $p_i \leq q_i$ for all i with $1 \leq i \leq n$.

Theorem 2.10.2 (Dickson) *Let $S \subseteq \mathbb{N}^n$, and let M be the set consisting of all elements of S that are minimal in the relation "is dominated by". Thus,*

$$M = \{q \in S : \textit{ there is no } p \textit{ in } S \setminus \{q\} \textit{ such that } p \textit{ is dominated by } q\}.$$

Then, the set M is finite.

We will prove this theorem by induction on the dimension n. If $n = 1$, then either $M = \emptyset$ (if $S = \emptyset$) or M consists of exactly one element (if $S \neq \emptyset$). Therefore, the theorem holds if $n = 1$. Let $n \geq 2$ and assume the theorem holds for all subsets of $\mathbb{N}^{n-1}$. Let S be a subset of $\mathbb{N}^n$ and consider the set M of minimal elements in S. If $S = \emptyset$, then $M = \emptyset$ and, thus, M is finite. Assume that $S \neq \emptyset$. We fix an arbitrary element q in M. If $p \in M \setminus \{q\}$, then q is not dominated by p. Therefore, there exists an index i such that $p_i \leq q_i - 1$. It follows that

$$M \setminus \{q\} \subseteq \bigcup_{i=1}^{n} \left(\mathbb{N}^{i-1} \times [1, q_i - 1] \times \mathbb{N}^{n-i}\right).$$

For all i and k with $1 \leq i \leq n$ and $1 \leq k \leq q_i - 1$, we define

$$S_{ik} = \{p \in S : p_i = k\}$$

and

$$M_{ik} = \{p \in M : p_i = k\}.$$

Then,

$$M \setminus \{q\} = \bigcup_{i=1}^{n} \bigcup_{k=1}^{q_i - 1} M_{ik}. \tag{2.7}$$

Lemma 2.10.3 *M_{ik} is a subset of the set of all elements of S_{ik} that are minimal in the relation "is dominated by".*

Proof. Let p be an element of M_{ik}, and assume that p is not minimal in S_{ik}. Then there is an element r in S_{ik}, such that $r \neq p$ and r is dominated by p. Since p and r are both elements of S, it follows that $p \notin M$. This is a contradiction. ∎

Since the set S_{ik} is basically a subset of $\mathbb{N}^{n-1}$, it follows from the induction hypothesis that S_{ik} contains finitely many minimal elements. This, combined with Lemma 2.10.3, implies that M_{ik} is a finite set. Thus, by (2.7), $M \setminus \{q\}$ is the union of finitely many finite sets. Therefore, the set M is finite.

2.10.2 Proof of Higman's Theorem

We give the proof of Theorem 2.10.1 for the case when $\Sigma = \{0, 1\}$. If $L = \emptyset$ or $SUBSEQ(L) = \{0, 1\}^*$, then $SUBSEQ(L)$ is obviously a regular language.

Hence, we may assume that L is non-empty and $SUBSEQ(L)$ is a proper subset of $\{0,1\}^*$.

We fix a string z of length at least two in the complement $\overline{SUBSEQ(L)}$ of the language $SUBSEQ(L)$. Observe that this is possible, because $\overline{SUBSEQ(L)}$ is an infinite language. We insert 0s and 1s into z, such that, in the resulting string z', 0s and 1s alternate. For example, if $z = 0011101011$, then $z' = 01010101010101$. Let $n = |z'| - 1$, where $|z'|$ denotes the length of z'. Then, $n \geq |z| - 1 \geq 1$.

A $(0,1)$-*alternation* in a binary string x is any occurrence of 01 or 10 in x. For example, the string 1101001 contains four $(0,1)$-alternations. We define

$$A = \{x \in \{0,1\}^* : \ x \text{ has at most } n \text{ many } (0,1)\text{-alternations}\}.$$

Lemma 2.10.4 *$SUBSEQ(L) \subseteq A$.*

Proof. Let $x \in SUBSEQ(L)$ and assume that $x \notin A$. Then, x has at least $n+1 = |z'|$ many $(0,1)$-alternations and, therefore, z' is a subsequence of x. In particular, z is a subsequence of x. Since $x \in SUBSEQ(L)$, it follows that $z \in SUBSEQ(L)$, which is a contradiction. ■

Lemma 2.10.5 $\overline{SUBSEQ(L)} = \left(A \cap \overline{SUBSEQ(L)}\right) \cup \overline{A}$.

Proof. Follows from Lemma 2.10.4. ■

Lemma 2.10.6 *The language $\overline{A}$ is regular.*

Proof. The complement $\overline{A}$ of A is the language consisting of all binary strings with at least $n+1$ many $(0,1)$-alternations. If, for example, $n = 3$, then $\overline{A}$ is described by the regular expression

$$(00^*11^*00^*11^*0(0\cup 1)^*) \ \cup \ (11^*00^*11^*00^*1(0\cup 1)^*)\,.$$

This should convince you that the claim is true for any value of n. ■

For any $b \in \{0,1\}$ and for any $k \geq 0$, we define A_{bk} to be the language consisting of all binary strings that start with a b and have exactly k many $(0,1)$-alternations. Then, we have

$$A = \{\epsilon\} \cup \left(\bigcup_{b=0}^{1} \bigcup_{k=0}^{n} A_{bk} \right).$$

Thus, if we define

$$F_{bk} = A_{bk} \cap \overline{SUBSEQ(L)},$$

and use the fact that $\epsilon \in SUBSEQ(L)$ (which is true because $L \neq \emptyset$), then

$$A \cap \overline{SUBSEQ(L)} = \bigcup_{b=0}^{1} \bigcup_{k=0}^{n} F_{bk}. \tag{2.8}$$

For any $b \in \{0,1\}$ and for any $k \geq 0$, consider the relation "is a subsequence of" on the language F_{bk}. We define M_{bk} to be the language consisting of all strings in F_{bk} that are minimal in this relation. Thus,

$$M_{bk} = \{x \in F_{bk} : \text{ there is no } x' \text{ in } F_{bk} \setminus \{x\} \text{ such that } x' \text{ is a subsequence of } x\}.$$

It is clear that

$$F_{bk} = \bigcup_{x \in M_{bk}} \{y \in F_{bk} : \ x \text{ is a subsequence of } y\}.$$

If $x \in M_{bk}$, $y \in A_{bk}$, and x is a subsequence of y, then y must be in $\overline{SUBSEQ(L)}$ and, therefore, y must be in F_{bk}. To prove this, assume that $y \in SUBSEQ(L)$. Then, $x \in SUBSEQ(L)$, contradicting the fact that $x \in M_{bk} \subseteq F_{bk} \subseteq \overline{SUBSEQ(L)}$. It follows that

$$F_{bk} = \bigcup_{x \in M_{bk}} \{y \in A_{bk} : \ x \text{ is a subsequence of } y\}. \tag{2.9}$$

Lemma 2.10.7 *Let $b \in \{0,1\}$ and $0 \leq k \leq n$, and let x be an element of M_{bk}. Then, the language*

$$\{y \in A_{bk} : \ x \textit{ is a subsequence of } y\}$$

is regular.

Proof. We will prove the claim by means of an example. Assume that $b = 1$, $k = 3$, and $x = 11110001000$. Then, the language

$$\{y \in A_{bk} : \ x \text{ is a subsequence of } y\}$$

is described by the regular expression

$$11111^*0000^*11^*0000^*.$$

This should convince you that the claim is true in general. ∎

Lemma 2.10.8 *For each $b \in \{0,1\}$ and each $0 \leq k \leq n$, the set M_{bk} is finite.*

Proof. Again, we will prove the claim by means of an example. Assume that $b = 1$ and $k = 3$. Any string in F_{bk} can be written as $1^a0^b1^c0^d$, for some integers $a, b, c, d \geq 1$. Consider the function $\varphi : F_{bk} \rightarrow \mathbb{N}^4$ that is defined by $\varphi(1^a0^b1^c0^d) = (a, b, c, d)$. Then, φ is an injective function, and the following is true, for any two strings x and x' in F_{bk}:

x is a subsequence of x' if and only if $\varphi(x)$ is dominated by $\varphi(x')$.

It follows that the elements of M_{bk} are in one-to-one correspondence with those elements of $\varphi(F_{bk})$ that are minimal in the relation "is dominated by". The lemma thus follows from Dickson's Theorem. ∎

Now we can complete the proof of Higman's Theorem:

- It follows from (2.9) and Lemmas 2.10.7 and 2.10.8, that F_{bk} is the union of finitely many regular languages. Therefore, by Theorem 2.3.1, F_{bk} is a regular language.
- It follows from (2.8) that $A \cap \overline{SUBSEQ(L)}$ is the union of finitely many regular languages. Therefore, again by Theorem 2.3.1, $A \cap \overline{SUBSEQ(L)}$ is a regular language.
- Since $A \cap \overline{SUBSEQ(L)}$ is regular and, by Lemma 2.10.6, $\overline{A}$ is regular, it follows from Lemma 2.10.5 that $\overline{SUBSEQ(L)}$ is the union of two regular languages. Therefore, by Theorem 2.3.1, $\overline{SUBSEQ(L)}$ is a regular language.
- Since $\overline{SUBSEQ(L)}$ is regular, it follows from Theorem 2.6.4 that the language $SUBSEQ(L)$ is regular as well.

Exercises

2.1 For each of the following languages, construct a DFA that accepts the language. In all cases, the alphabet is $\{0, 1\}$.

1. $\{w :$ the length of w is divisible by three$\}$

2. $\{w : $ 110 is not a substring of $w\}$

3. $\{w : w$ contains at least five 1s$\}$

4. $\{w : w$ contains the substring 1011$\}$

5. $\{w : w$ contains at least two 1s and at most two 0s$\}$

6. $\{w : w$ contains an odd number of 1s or exactly two 0s$\}$

7. $\{w : w$ begins with 1 and ends with 0$\}$

8. $\{w : $ every odd position in w is 1$\}$

9. $\{w : w$ has length at least 3 and its third symbol is 0$\}$

10. $\{\epsilon, 0\}$

2.2 For each of the following languages, construct an NFA, with the specified number of states, that accepts the language. In all cases, the alphabet is $\{0, 1\}$.

1. The language $\{w : w$ ends with 10$\}$ with three states.

2. The language $\{w : w$ contains the substring 1011$\}$ with five states.

3. The language $\{w : w$ contains an odd number of 1s or exactly two 0s$\}$ with six states.

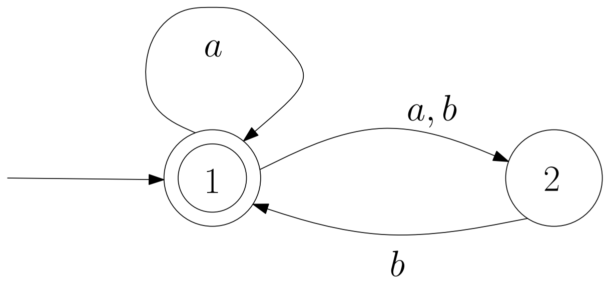
2.3 For each of the following languages, construct an NFA that accepts the language. In all cases, the alphabet is $\{0, 1\}$.

1. $\{w : w$ contains the substring 11001$\}$

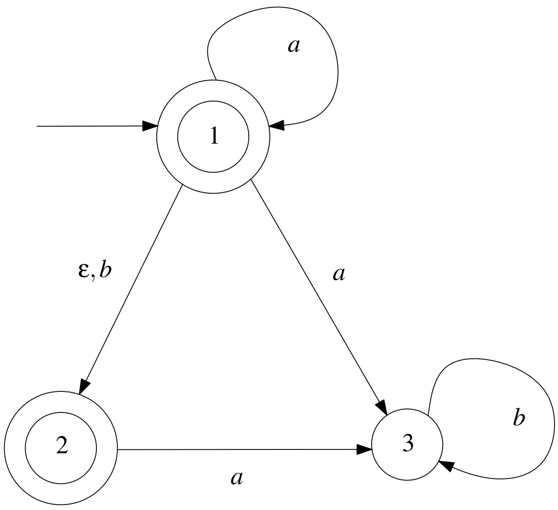
2. $\{w : w$ has length at least 2 and does not end with 10$\}$

3. $\{w : w$ begins with 1 or ends with 0$\}$

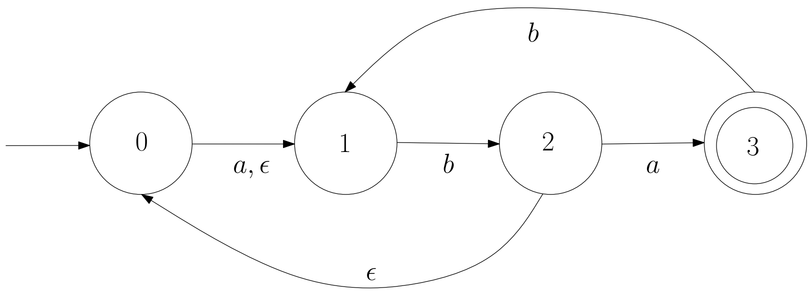
2.4 Convert the following NFA to an equivalent DFA.



2.5 Convert the following NFA to an equivalent DFA.



2.6 Convert the following NFA to an equivalent DFA.



2.7 In the proof of Theorem 2.6.3, we introduced a new start state q_0, which is also an accept state. Explain why the following is not a valid proof of Theorem 2.6.3:

Let $N = (Q_1, \Sigma, \delta_1, q_1, F_1)$ be an NFA, such that $A = L(N)$. Define the NFA $M = (Q_1, \Sigma, \delta, q_1, F)$, where

1. $F = \{q_1\} \cup F_1$.

2. $\delta : Q_1 \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q_1)$ is defined as follows: For any $r \in Q_1$ and for any $a \in \Sigma_\epsilon$,

$$\delta(r,a) = \begin{cases} \delta_1(r,a) & \text{if } r \in Q_1 \text{ and } r \notin F_1, \\ \delta_1(r,a) & \text{if } r \in F_1 \text{ and } a \neq \epsilon, \\ \delta_1(r,a) \cup \{q_1\} & \text{if } r \in F_1 \text{ and } a = \epsilon. \end{cases}$$

Then $L(M) = A^*$.

2.8 Prove Theorem 2.6.4.

2.9 Let A be a language over the alphabet $\Sigma = \{0,1\}$ and let $\overline{A}$ be the *complement* of A. Thus, $\overline{A}$ is the language consisting of all binary strings that are not in A.

Assume that A is a regular language. Let $M = (Q, \Sigma, \delta, q, F)$ be a nondeterministic finite automaton (NFA) that accepts A.

Consider the NFA $N = (Q, \Sigma, \delta, q, \overline{F})$, where $\overline{F} = Q \setminus F$ is the complement of F. Thus, N is obtained from M by turning all accept states into nonaccept states, and turning all nonaccept states into accept states.

1. Is it true that the language accepted by N is equal to $\overline{A}$?

2. Assume now that M is a deterministic finite automaton (DFA) that accepts A. Define N as above; thus, turn all accept states into nonaccept states, and turn all nonaccept states into accept states. Is it true that the language accepted by N is equal to $\overline{A}$?

2.10 Recall the alternative definition for the star of a language A that we gave just before Theorem 2.3.1.

In Theorems 2.3.1 and 2.6.2, we have shown that the class of regular languages is closed under the union and concatenation operations. Since $A^* = \bigcup_{k=0}^{\infty} A^k$, why doesn't this imply that the class of regular languages is closed under the star operation?

2.11 Let A and B be two regular languages over the same alphabet Σ. Prove that the difference of A and B, i.e., the language

$$A \setminus B = \{w : w \in A \text{ and } w \notin B\}$$

is a regular language.

2.12 For each of the following regular expressions, give two strings that are members and two strings that are not members of the language described by the expression. The alphabet is $\Sigma = \{a, b\}$.

1. $a(ba)^*b$.
2. $(a \cup b)^*a(a \cup b)^*b(a \cup b)^*a(a \cup b)^*$.
3. $(a \cup ba \cup bb)(a \cup b)^*$.

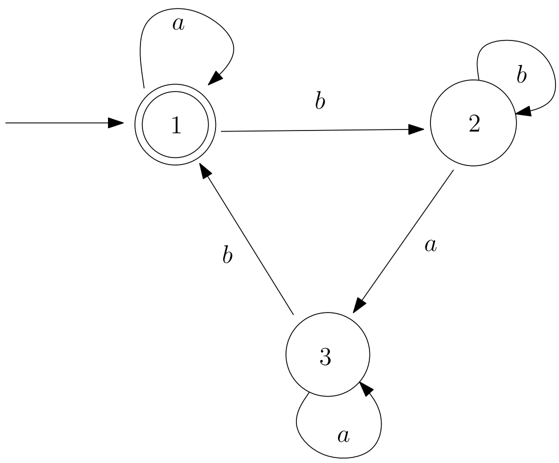
2.13 Give regular expressions describing the following languages. In all cases, the alphabet is $\{0, 1\}$.

1. $\{w : w$ contains at least three 1s$\}$.
2. $\{w : w$ contains at least two 1s and at most one 0$\}$,
3. $\{w : w$ contains an even number of 0s and exactly two 1s$\}$.
4. $\{w : w$ contains exactly two 0s and at least two 1s$\}$.
5. $\{w : w$ contains an even number of 0s and each 0 is followed by at least one 1$\}$.
6. $\{w :$ every odd position in w is 1$\}$.

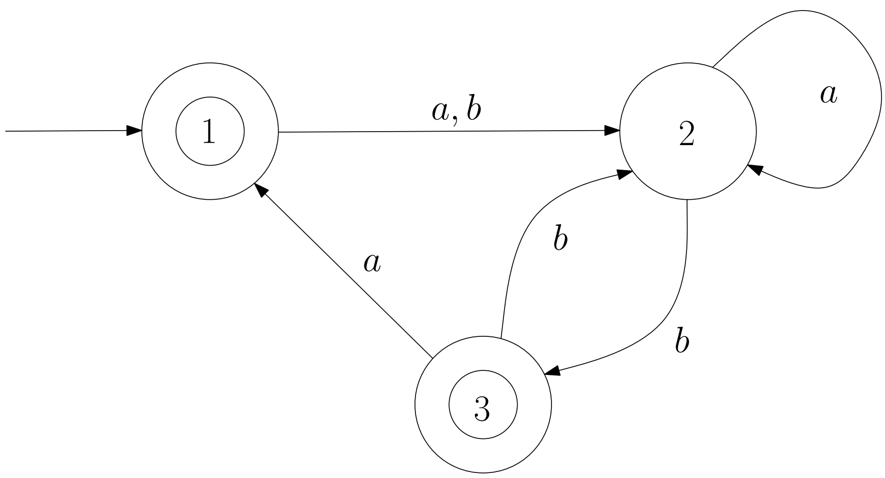
2.14 Convert each of the following regular expressions to an NFA.

1. $(0 \cup 1)^*000(0 \cup 1)^*$
2. $(((10)^*(00)) \cup 10)^*$
3. $((0 \cup 1)(11)^* \cup 0)^*$

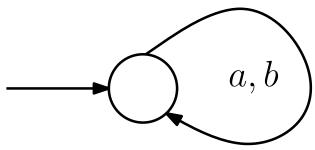
2.15 Convert the following DFA to a regular expression.



2.16 Convert the following DFA to a regular expression.



2.17 Convert the following DFA to a regular expression.



2.18 1. Let A be a non-empty regular language. Prove that there exists an NFA that accepts A and that has exactly one accept state.

2. For any string $w = w_1 w_2 \ldots w_n$, we denote by w^R the string obtained by reading w backwards, i.e., $w^R = w_n w_{n-1} \ldots w_2 w_1$. For any language A, we define A^R to be the language obtained by reading all strings in A backwards, i.e.,
$$A^R = \{w^R : w \in A\}.$$
Let A be a non-empty regular language. Prove that the language A^R is also regular.

2.19 If $n \geq 1$ is an integer and $w = a_1 a_2 \ldots a_n$ is a string, then for any i with $0 \leq i < n$, the string $a_1 a_2 \ldots a_i$ is called a *proper prefix* of w. (If $i = 0$, then $a_1 a_2 \ldots a_i = \epsilon$.)

For any language L, we define $\text{MIN}(L)$ to be the language

$$\text{MIN}(L) = \{w \in L : \text{ no proper prefix of } w \text{ belongs to } L\}.$$

Prove the following claim: If L is a regular language, then $\text{MIN}(L)$ is regular as well.

2.20 Use the pumping lemma to prove that the following languages are not regular.

1. $\{a^n b^m c^{n+m} : n \geq 0, m \geq 0\}$.
2. $\{a^n b^n c^{2n} : n \geq 0\}$.
3. $\{a^n b^m a^n : n \geq 0, m \geq 0\}$.
4. $\{a^{2^n} : n \geq 0\}$. (Remark: a^{2^n} is the string consisting of 2^n many a's.)
5. $\{a^n b^m c^k : n \geq 0, m \geq 0, k \geq 0, n^2 + m^2 = k^2\}$.
6. $\{uvu : u \in \{a,b\}^*, u \neq \epsilon, v \in \{a,b\}^*\}$.

2.21 Prove that the language

$$\{a^m b^n : m \geq 0, n \geq 0, m \neq n\}$$

is not regular. (Using the pumping lemma for this one is a bit tricky. You can avoid using the pumping lemma by combining results about the closure under regular operations.)

2.22 1. Give an example of a regular language A and a non-regular language B for which $A \subseteq B$.

2. Give an example of a non-regular language A and a regular language B for which $A \subseteq B$.

2.23 Let A be a language consisting of finitely many strings.

1. Prove that A is a regular language.

2. Let n be the maximum length of any string in A. Prove that *every* deterministic finite automaton (DFA) that accepts A has at least $n+1$ states. (*Hint:* How is the pumping length chosen in the proof of the pumping lemma?)

2.24 Let L be a regular language, let M be a DFA whose language is equal to L, and let p be the number of states of M. Prove that $L \neq \emptyset$ if and only if L contains a string of length less than p.

2.25 Let L be a regular language, let M be a DFA whose language is equal to L, and let p be the number of states of M. Prove that L is an infinite language if and only if L contains a string w with $p \leq |w| \leq 2p - 1$.

2.26 Let Σ be a non-empty alphabet, and let L be a language over Σ, i.e., $L \subseteq \Sigma^*$. We define a binary relation R_L on $\Sigma^* \times \Sigma^*$, in the following way: For any two strings u and u' in Σ^*,

$$uR_Lu' \text{ if and only if } (\forall v \in \Sigma^* : uv \in L \Leftrightarrow u'v \in L) .$$

Prove that R_L is an equivalence relation.

2.27 Let $\Sigma = \{0, 1\}$, let

$$L = \{w \in \Sigma^* : |w| \text{ is odd}\},$$

and consider the relation R_L defined in Exercise 2.26.

1. Prove that for any two strings u and u' in Σ^*,

$$uR_Lu' \Leftrightarrow |u| - |u'| \text{ is even}.$$

2. Determine all equivalence classes of the relation R_L.

2.28 Let Σ be a non-empty alphabet, and let L be a language over Σ, i.e., $L \subseteq \Sigma^*$. Recall the equivalence relation R_L that was defined in Exercise 2.26.

1. Assume that L is a regular language, and let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that accepts L. Let u and u' be strings in Σ^*. Let q be the state reached, when following the path in the state diagram of M, that starts in q_0 and that is obtained by reading the string u. Similarly, let q' be the state reached, when following the path in the state diagram of M, that starts in q_0 and that is obtained by reading the string u'.

   Prove the following: If $q = q'$, then $u R_L u'$.

2. Prove the following claim: If L is a regular language, then the equivalence relation R_L has a finite number of equivalence classes.

2.29 Let L be the language defined by

$$L = \{uu^R : u \in \{0,1\}^*\}.$$

In words, a string is in L if and only if its length is even, and the second half is the reverse of the first half. Consider the equivalence relation R_L that was defined in Exercise 2.26.

1. Let m and n be two distinct positive integers and consider the two strings $u = 0^m 1$ and $u' = 0^n 1$. Prove that $\neg(u R_L u')$.

2. Prove that L is not a regular language, without using the pumping lemma.

3. Use the pumping lemma to prove that L is not a regular language.

2.30 In this exercise, we will show that the converse of the pumping lemma does, in general, not hold. Consider the language

$$A = \{a^m b^n c^n : m \geq 1, n \geq 0\} \cup \{b^n c^k : n \geq 0, k \geq 0\}.$$

1. Show that A satisfies the conclusion of the pumping lemma for $p = 1$. Thus, show that every string s in A whose length is at least p can be written as $s = xyz$, such that $y \neq \epsilon$, $|xy| \leq p$, and $xy^i z \in A$ for all $i \geq 0$.

2. Consider the equivalence relation R_A that was defined in Exercise 2.26. Let n and n' be two distinct non-negative integers and consider the two strings $u = ab^n$ and $u' = ab^{n'}$. Prove that $\neg(uR_Au')$.

3. Prove that A is not a regular language.

Chapter 3

Context-Free Languages

In this chapter, we introduce the class of context-free languages. As we will see, this class contains all regular languages, as well as some nonregular languages such as $\{0^n 1^n : n \geq 0\}$.

The class of context-free languages consists of languages that have some sort of recursive structure. We will see two equivalent methods to obtain this class. We start with context-free grammars, which are used for defining the syntax of programming languages and their compilation. Then we introduce the notion of (nondeterministic) pushdown automata, and show that these automata have the same power as context-free grammars.

3.1 Context-free grammars

We start with an example. Consider the following five (substitution) rules:

$$\begin{array}{lcl} S & \rightarrow & AB \\ A & \rightarrow & a \\ A & \rightarrow & aA \\ B & \rightarrow & b \\ B & \rightarrow & bB \end{array}$$

Here, S, A, and B are *variables*, S is the *start variable*, and a and b are *terminals*. We use these rules to derive strings consisting of terminals (i.e., elements of $\{a, b\}^*$), in the following manner:

1. Initialize the *current string* to be the string consisting of the start variable S.

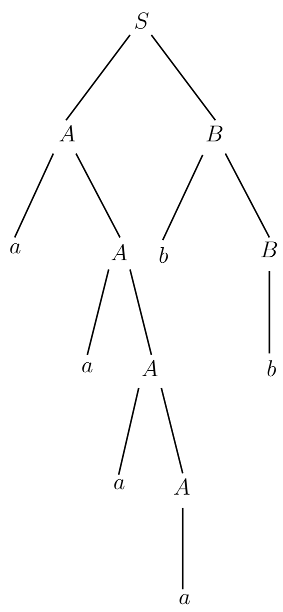
2. Take any variable in the current string and take any rule that has this variable on the left-hand side. Then, in the current string, replace this variable by the right-hand side of the rule.

3. Repeat 2. until the current string only contains terminals.

For example, the string *aaaabb* can be derived in the following way:

$$\begin{aligned} S &\Rightarrow AB \\ &\Rightarrow aAB \\ &\Rightarrow aAbB \\ &\Rightarrow aaAbB \\ &\Rightarrow aaaAbB \\ &\Rightarrow aaaabB \\ &\Rightarrow aaaabb \end{aligned}$$

This derivation can also be represented using a *parse tree*, as in the figure below:



The five rules in this example constitute a context-free grammar. The language of this grammar is the set of all strings that

- can be derived from the start variable and
- only contain terminals.

For this example, the language is

$$\{a^m b^n : m \geq 1, n \geq 1\},$$

because every string of the form $a^m b^n$, for some $m \geq 1$ and $n \geq 1$, can be derived from the start variable, whereas no other string over the alphabet $\{a, b\}$ can be derived from the start variable.

Definition 3.1.1 A context-free grammar is a 4-tuple $G = (V, \Sigma, R, S)$, where

1. V is a finite set, whose elements are called *variables*,
2. Σ is a finite set, whose elements are called *terminals*,
3. $V \cap \Sigma = \emptyset$,
4. S is an element of V; it is called the *start variable*,
5. R is a finite set, whose elements are called *rules*. Each rule has the form $A \to w$, where $A \in V$ and $w \in (V \cup \Sigma)^*$.

In our example, we have $V = \{S, A, B\}$, $\Sigma = \{a, b\}$, and

$$R = \{S \to AB, A \to a, A \to aA, B \to b, B \to bB\}.$$

Definition 3.1.2 Let $G = (V, \Sigma, R, S)$ be a context-free grammar. Let A be an element in V and let u, v, and w be strings in $(V \cup \Sigma)^*$ such that $A \to w$ is a rule in R. We say that the string uwv can be *derived in one step from* the string uAv, and write this as

$$uAv \Rightarrow uwv.$$

In other words, by applying the rule $A \to w$ to the string uAv, we obtain the string uwv. In our example, we see that $aaAbb \Rightarrow aaaAbb$.

Definition 3.1.3 Let $G = (V, \Sigma, R, S)$ be a context-free grammar. Let u and v be strings in $(V \cup \Sigma)^*$. We say that v can be *derived from* u, and write this as $u \overset{*}{\Rightarrow} v$, if one of the following two conditions holds:

1. $u = v$ or
2. there exist an integer $k \geq 2$ and a sequence $u_1, u_2, \ldots, u_k$ of strings in $(V \cup \Sigma)^*$, such that
   (a) $u = u_1$,
   (b) $v = u_k$, and
   (c) $u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k$.

In other words, by starting with the string u and applying rules zero or more times, we obtain the string v. In our example, we see that $aaAbB \stackrel{*}{\Rightarrow} aaaabbbB$.

Definition 3.1.4 Let $G = (V, \Sigma, R, S)$ be a context-free grammar. The *language* of G is defined to be the set of all strings in Σ^* that can be derived from the start variable S:

$$L(G) = \{w \in \Sigma^* : S \stackrel{*}{\Rightarrow} w\}.$$

Definition 3.1.5 A language L is called *context-free*, if there exists a context-free grammar G such that $L(G) = L$.

3.2 Examples of context-free grammars

3.2.1 Properly nested parentheses

Consider the context-free grammar $G = (V, \Sigma, R, S)$, where $V = \{S\}$, $\Sigma = \{a, b\}$, and

$$R = \{S \rightarrow \epsilon, S \rightarrow aSb, S \rightarrow SS\}.$$

We write the three rules in R as

$$S \rightarrow \epsilon | aSb | SS,$$

where you can think of "|" as being a short-hand for "or".

By applying the rules in R, starting with the start variable S, we obtain, for example,

$$\begin{aligned} S &\Rightarrow SS \\ &\Rightarrow aSbS \\ &\Rightarrow aSbSS \\ &\Rightarrow aSSbSS \\ &\Rightarrow aaSbSbSS \\ &\Rightarrow aabSbSS \\ &\Rightarrow aabbSS \\ &\Rightarrow aabbaSbS \\ &\Rightarrow aabbabS \\ &\Rightarrow aabbabaSb \\ &\Rightarrow aabbabab \end{aligned}$$

What is the language $L(G)$ of this context-free grammar G? If we think of a as being a left-parenthesis "(", and of b as being a right-parenthesis ")", then $L(G)$ is the language consisting of all strings of properly nested parentheses. Here is the explanation: Any string of properly nested parentheses is either

- empty (which we derive from S by the rule $S \to \epsilon$),
- consists of a left-parenthesis, followed by an arbitrary string of properly nested parentheses, followed by a right-parenthesis (these are derived from S by first applying the rule $S \to aSb$), or
- consists of an arbitrary string of properly nested parentheses, followed by an arbitrary string of properly nested parentheses (these are derived from S by first applying the rule $S \to SS$).

3.2.2 A context-free grammar for a nonregular language

Consider the language $L_1 = \{0^n 1^n : n \geq 0\}$. We have seen in Section 2.9.1 that L_1 is not a regular language. We claim that L_1 is a context-free language.

In order to prove this claim, we have to construct a context-free grammar G_1 such that $L(G_1) = L_1$.

Observe that any string in L_1 is either

- empty or
- consists of a 0, followed by an arbitrary string in L_1, followed by a 1.

This leads to the context-free grammar $G_1 = (V_1, \Sigma, R_1, S_1)$, where $V_1 = \{S_1\}$, $\Sigma = \{0, 1\}$, and R_1 consists of the rules

$$S_1 \rightarrow \epsilon | 0S_1 1.$$

Hence, $R_1 = \{S_1 \rightarrow \epsilon, S_1 \rightarrow 0S_1 1\}$.

To derive the string $0^n 1^n$ from the start variable S_1, we do the following:

- Starting with S_1, apply the rule $S_1 \rightarrow 0S_1 1$ exactly n times. This gives the string $0^n S_1 1^n$.
- Apply the rule $S_1 \rightarrow \epsilon$. This gives the string $0^n 1^n$.

It is not difficult to see that these are the only strings that can be derived from the start variable S_1. Thus, $L(G_1) = L_1$.

In a symmetric way, we see that the context-free grammar $G_2 = (V_2, \Sigma, R_2, S_2)$, where $V_2 = \{S_2\}$, $\Sigma = \{0, 1\}$, and R_2 consists of the rules

$$S_2 \rightarrow \epsilon | 1S_2 0,$$

has the property that $L(G_2) = L_2$, where $L_2 = \{1^n 0^n : n \geq 0\}$. Thus, L_2 is a context-free language.

Define $L = L_1 \cup L_2$, i.e.,

$$L = \{0^n 1^n : n \geq 0\} \cup \{1^n 0^n : n \geq 0\}.$$

The context-free grammar $G = (V, \Sigma, R, S)$, where $V = \{S, S_1, S_2\}$, $\Sigma = \{0, 1\}$, and R consists of the rules

$$\begin{array}{lcl} S & \rightarrow & S_1 | S_2 \\ S_1 & \rightarrow & \epsilon | 0S_1 1 \\ S_2 & \rightarrow & \epsilon | 1S_2 0, \end{array}$$

has the property that $L(G) = L$. Hence, L is a context-free language.

3.2.3 A context-free grammar for the complement of a nonregular language

Let L be the (nonregular) language $L = \{0^n 1^n : n \geq 0\}$. We want to prove that the complement $\overline{L}$ of L is a context-free language. Hence, we want to construct a context-free grammar G whose language is equal to $\overline{L}$. Observe that a binary string w is in $\overline{L}$ if and only if

1. $w = 0^m 1^n$, for some integers m and n with $0 \leq m < n$, or
2. $w = 0^m 1^n$, for some integers m and n with $0 \leq n < m$, or
3. w contains 10 as a substring.

Thus, we can write $\overline{L}$ as the union of the languages of all strings of type 1., type 2., and type 3.

Any string of type 1. is either

- the string 1,
- consists of a string of type 1., followed by one 1, or
- consists of one 0, followed by an arbitrary string of type 1., followed by one 1.

Thus, using the rules

$$S_1 \to 1|S_1 1|0S_1 1,$$

we can derive, from S_1, all strings of type 1.

Similarly, using the rules

$$S_2 \to 0|0S_2|0S_2 1,$$

we can derive, from S_2, all strings of type 2.

Any string of type 3.

- consists of an arbitrary binary string, followed by the string 10, followed by an arbitrary binary string.

Using the rules

$$X \to \epsilon|0X|1X,$$

we can derive, from X, all binary strings. Thus, by combining these with the rule

$$S_3 \rightarrow X10X,$$

we can derive, from S_3, all strings of type 3.

We arrive at the context-free grammar $G = (V, \Sigma, R, S)$, where $V = \{S, S_1, S_2, S_3, X\}$, $\Sigma = \{0, 1\}$, and R consists of the rules

$$\begin{array}{lcl} S & \rightarrow & S_1|S_2|S_3 \\ S_1 & \rightarrow & 1|S_11|0S_11 \\ S_2 & \rightarrow & 0|0S_2|0S_21 \\ S_3 & \rightarrow & X10X \\ X & \rightarrow & \epsilon|0X|1X \end{array}$$

To summarize, we have

$$S_1 \stackrel{*}{\Rightarrow} 0^m1^n, \text{ for all integers } m \text{ and } n \text{ with } 0 \leq m < n,$$

$$S_2 \stackrel{*}{\Rightarrow} 0^m1^n, \text{ for all integers } m \text{ and } n \text{ with } 0 \leq n < m,$$

$$X \stackrel{*}{\Rightarrow} u, \text{ for each string } u \text{ in } \{0,1\}^*,$$

and

$$S_3 \stackrel{*}{\Rightarrow} w, \text{ for every binary string } w \text{ that contains 10 as a substring.}$$

From these observations, it follows that that $L(G) = \overline{L}$.

3.2.4 A context-free grammar that verifies addition

Consider the language

$$L = \{a^nb^mc^{n+m} : n \geq 0, m \geq 0\}.$$

Using the pumping lemma for regular languages (Theorem 2.9.1), it can be shown that L is not a regular language. We will construct a context-free grammar G whose language is equal to L, thereby proving that L is a context-free language.

First observe that $\epsilon \in L$. Therefore, we will take $S \rightarrow \epsilon$ to be one of the rules in the grammar.

Let us see how we can derive all strings in L from the start variable S:

1. Every time we add an a, we also add a c. In this way, we obtain all strings of the form $a^n c^n$, where $n \geq 0$.

2. Given a string of the form $a^n c^n$, we start adding bs. Every time we add a b, we also add a c. Observe that every b has to be added between the as and the cs. Therefore, we use a variable B as a "pointer" to the position in the current string where a b can be added: Instead of deriving $a^n c^n$ from S, we derive the string $a^n B c^n$. Then, from B, we derive all strings of the form $b^m c^m$, where $m \geq 0$.

We obtain the context-free grammar $G = (V, \Sigma, R, S)$, where $V = \{S, A, B\}$, $\Sigma = \{a, b, c\}$, and R consists of the rules

$$\begin{array}{lcl} S & \rightarrow & \epsilon | A \\ A & \rightarrow & \epsilon | aAc | B \\ B & \rightarrow & \epsilon | bBc \end{array}$$

The facts that

- $A \stackrel{*}{\Rightarrow} a^n B c^n$, for every $n \geq 0$,
- $B \stackrel{*}{\Rightarrow} b^m c^m$, for every $m \geq 0$,

imply that the following strings can be derived from the start variable S:

- $S \stackrel{*}{\Rightarrow} a^n B c^n \stackrel{*}{\Rightarrow} a^n b^m c^m c^n = a^n b^m c^{n+m}$, for all $n \geq 0$ and $m \geq 0$.

In fact, no other strings in $\{a, b, c\}^*$ can be derived from S. Therefore, we have $L(G) = L$. Since

$$S \Rightarrow A \Rightarrow B \Rightarrow \epsilon,$$

we can simplify this grammar G, by eliminating the rules $S \rightarrow \epsilon$ and $A \rightarrow \epsilon$. This gives the context-free grammar $G' = (V, \Sigma, R', S)$, where $V = \{S, A, B\}$, $\Sigma = \{a, b, c\}$, and R' consists of the rules

$$\begin{array}{lcl} S & \rightarrow & A \\ A & \rightarrow & aAc | B \\ B & \rightarrow & \epsilon | bBc \end{array}$$

Finally, observe that we do not need S; instead, we can use A as start variable. This gives our final context-free grammar $G'' = (V, \Sigma, R'', A)$, where $V = \{A, B\}$, $\Sigma = \{a, b, c\}$, and R'' consists of the rules

$$\begin{array}{lcl} A & \rightarrow & aAc | B \\ B & \rightarrow & \epsilon | bBc \end{array}$$

3.3 Regular languages are context-free

We mentioned already that the class of context-free languages includes the class of regular languages. In this section, we will prove this claim.

Theorem 3.3.1 *Let Σ be an alphabet and let $L \subseteq \Sigma^*$ be a regular language. Then L is a context-free language.*

Proof. Since L is a regular language, there exists a deterministic finite automaton $M = (Q, \Sigma, \delta, q, F)$ that accepts L.

To prove that L is context-free, we have to define a context-free grammar $G = (V, \Sigma, R, S)$, such that $L = L(M) = L(G)$. Thus, G must have the following property: For every string $w \in \Sigma^*$,

$$w \in L(M) \text{ if and only if } w \in L(G),$$

which can be reformulated as

$$M \text{ accepts } w \text{ if and only if } S \stackrel{*}{\Rightarrow} w.$$

We will *define* the context-free grammar G in such a way that the following correspondence holds for any string $w = w_1 w_2 \ldots w_n$:

- Assume that M is in state A just after it has read the substring $w_1 w_2 \ldots w_i$.
- Then in the context-free grammar G, we have $S \stackrel{*}{\Rightarrow} w_1 w_2 \ldots w_i A$.

In the next step, M reads the symbol w_{i+1} and switches from state A to, say, state B; thus, $\delta(A, w_{i+1}) = B$. In order to guarantee that the above correspondence still holds, we have to add the rule $A \to w_{i+1} B$ to G.

Consider the moment when M has read the entire string w. Let A be the state M is in at that moment. By the above correspondence, we have

$$S \stackrel{*}{\Rightarrow} w_1 w_2 \ldots w_n A = wA.$$

Recall that G must have the property that

$$M \text{ accepts } w \text{ if and only if } S \stackrel{*}{\Rightarrow} w,$$

which is equivalent to

$$A \in F \text{ if and only if } S \stackrel{*}{\Rightarrow} w.$$

We guarantee this property by adding to G the rule $A \to \epsilon$ for every accept state A of M.

We are now ready to give the formal definition of the context-free grammar $G = (V, \Sigma, R, S)$:

- $V = Q$, i.e., the variables of G are the states of M.
- $S = q$, i.e., the start variable of G is the start state of M.
- R consists of the rules

  $$A \to aB, \text{ where } A \in Q,\ a \in \Sigma,\ B \in Q, \text{ and } \delta(A, a) = B,$$

  and

  $$A \to \epsilon, \text{ where } A \in F.$$

In words,

- every transition $\delta(A, a) = B$ of M (i.e., when M is in the state A and reads the symbol a, it switches to the state B) corresponds to a rule $A \to aB$ in the grammar G,
- every accept state A of M corresponds to a rule $A \to \epsilon$ in the grammar G.

We claim that $L(G) = L$. In order to prove this, we have to show that $L(G) \subseteq L$ and $L \subseteq L(G)$.

We prove that $L \subseteq L(G)$. Let $w = w_1 w_2 \ldots w_n$ be an arbitrary string in L. When the finite automaton M reads the string w, it visits the states $r_0, r_1, \ldots, r_n$, where

- $r_0 = q$, and
- $r_{i+1} = \delta(r_i, w_{i+1})$ for $i = 0, 1, \ldots, n-1$.

Since $w \in L = L(M)$, we know that $r_n \in F$.

It follows from the way we defined the grammar G that

- for each $i = 0, 1, \ldots, n-1$, $r_i \to w_{i+1} r_{i+1}$ is a rule in R, and
- $r_n \to \epsilon$ is a rule in R.

Therefore, we have

$$S = q = r_0 \Rightarrow w_1 r_1 \Rightarrow w_1 w_2 r_2 \Rightarrow \ldots \Rightarrow w_1 w_2 \ldots w_n r_n \Rightarrow w_1 w_2 \ldots w_n = w.$$

This proves that $w \in L(G)$.

The proof of the claim that $L(G) \subseteq L$ is left as an exercise. ∎

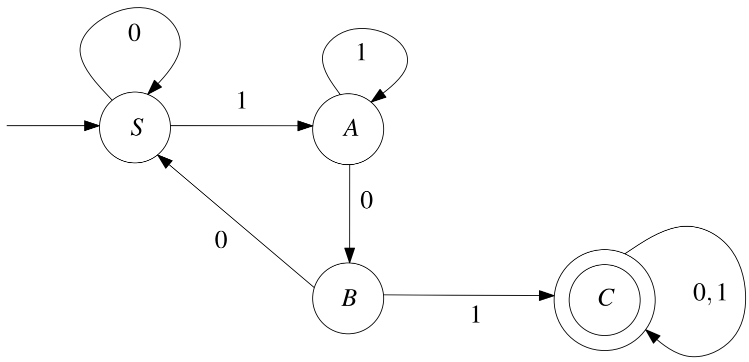
In Sections 2.9.1 and 3.2.2, we have seen that the language $\{0^n 1^n : n \geq 0\}$ is not regular, but context-free. Therefore, the class of all context-free languages properly contains the class of regular languages.

3.3.1 An example

Let L be the language defined as

$$L = \{w \in \{0,1\}^* : \text{ 101 is a substring of } w\}.$$

In Section 2.2.2, we have seen that L is a regular language. In that section, we constructed the following deterministic finite automaton M that accepts L (we have renamed the states):



We apply the construction given in the proof of Theorem 3.3.1 to convert M to a context-free grammar G whose language is equal to L. According to this construction, we have $G = (V, \Sigma, R, S)$, where $V = \{S, A, B, C\}$, $\Sigma = \{0, 1\}$, the start variable S is the start state of M, and R consists of the rules

$$\begin{array}{lcl} S & \rightarrow & 0S|1A \\ A & \rightarrow & 0B|1A \\ B & \rightarrow & 0S|1C \\ C & \rightarrow & 0C|1C|\epsilon \end{array}$$

Consider the string 010011011, which is an element of L. When the finite automaton M reads this string, it visits the states

$$S, S, A, B, S, A, A, B, C, C.$$

In the grammar G, this corresponds to the derivation

$$\begin{aligned}
S &\Rightarrow 0S \\
&\Rightarrow 01A \\
&\Rightarrow 010B \\
&\Rightarrow 0100S \\
&\Rightarrow 01001A \\
&\Rightarrow 010011A \\
&\Rightarrow 0100110B \\
&\Rightarrow 01001101C \\
&\Rightarrow 010011011C \\
&\Rightarrow 010011011.
\end{aligned}$$

Hence,

$$S \stackrel{*}{\Rightarrow} 010011011,$$

implying that the string 010011011 is in the language $L(G)$ of the context-free grammar G.

The string 10011 is not in the language L. When the finite automaton M reads this string, it visits the states

$$S, A, B, S, A, A,$$

i.e., after the string has been read, M is in the non-accept state A. In the grammar G, reading the string 10011 corresponds to the derivation

$$\begin{aligned}
S &\Rightarrow 1A \\
&\Rightarrow 10B \\
&\Rightarrow 100S \\
&\Rightarrow 1001A \\
&\Rightarrow 10011A.
\end{aligned}$$

Since A is not an accept state in M, the grammar G does not contain the rule $A \rightarrow \epsilon$. This implies that the string 10011 cannot be derived from the start variable S. Thus, 10011 is not in the language $L(G)$ of G.

3.4 Chomsky normal form

The rules in a context-free grammar $G = (V, \Sigma, R, S)$ are of the form

$$A \to w,$$

where A is a variable and w is a string over the alphabet $V \cup \Sigma$. In this section, we show that every context-free grammar G can be converted to a context-free grammar G', such that $L(G) = L(G')$, and the rules of G' are of a restricted form, as specified in the following definition:

Definition 3.4.1 A context-free grammar $G = (V, \Sigma, R, S)$ is said to be in *Chomsky normal form*, if every rule in R has one of the following three forms:

1. $A \to BC$, where A, B, and C are elements of V, $B \neq S$, and $C \neq S$.
2. $A \to a$, where A is an element of V and a is an element of Σ.
3. $S \to \epsilon$, where S is the start variable.

You should convince yourself that, for such a grammar, R contains the rule $S \to \epsilon$ if and only if $\epsilon \in L(G)$.

Theorem 3.4.2 *Let Σ be an alphabet and let $L \subseteq \Sigma^*$ be a context-free language. There exists a context-free grammar in Chomsky normal form, whose language is L.*

Proof. Since L is a context-free language, there exists a context-free grammar $G = (V, \Sigma, R, S)$, such that $L(G) = L$. We will transform G into a grammar that is in Chomsky normal form and whose language is equal to $L(G)$. The transformation consists of five steps.

Step 1: Eliminate the start variable from the right-hand side of the rules.

We define $G_1 = (V_1, \Sigma, R_1, S_1)$, where S_1 is the start variable (which is a new variable), $V_1 = V \cup \{S_1\}$, and $R_1 = R \cup \{S_1 \to S\}$. This grammar has the property that

- the start variable S_1 does not occur on the right-hand side of any rule in R_1, and
- $L(G_1) = L(G)$.

Step 2: An ϵ-*rule* is a rule that is of the form $A \to \epsilon$, where A is a variable that is not equal to the start variable. In the second step, we eliminate all ϵ-rules from G_1.

We consider all ϵ-rules, one after another. Let $A \to \epsilon$ be one such rule, where $A \in V_1$ and $A \neq S_1$. We modify G_1 as follows:

1. Remove the rule $A \to \epsilon$ from the current set R_1.

2. For each rule in the current set R_1 that is of the form

   (a) $B \to A$, add the rule $B \to \epsilon$ to R_1, unless this rule has already been deleted from R_1; observe that in this way, we replace the two-step derivation $B \Rightarrow A \Rightarrow \epsilon$ by the one-step derivation $B \Rightarrow \epsilon$;

   (b) $B \to uAv$ (where u and v are strings that are not both empty), add the rule $B \to uv$ to R_1; observe that in this way, we replace the two-step derivation $B \Rightarrow uAv \Rightarrow uv$ by the one-step derivation $B \Rightarrow uv$;

   (c) $B \to uAvAw$ (where u, v, and w are strings), add the rules $B \to uvw$, $B \to uAvw$, and $B \to uvAw$ to R_1; if $u = v = w = \epsilon$ and the rule $B \to \epsilon$ has already been deleted from R_1, then we do not add the rule $B \to \epsilon$;

   (d) treat rules in which A occurs more than twice on the right-hand side in a similar fashion.

We repeat this process until all ϵ-rules have been eliminated. Let R_2 be the set of rules, after all ϵ-rules have been eliminated. We define $G_2 = (V_2, \Sigma, R_2, S_2)$, where $V_2 = V_1$ and $S_2 = S_1$. This grammar has the property that

- the start variable S_2 does not occur on the right-hand side of any rule in R_2,

- R_2 does not contain any ϵ-rule (it may contain the rule $S_2 \to \epsilon$), and

- $L(G_2) = L(G_1) = L(G)$.

Step 3: A *unit-rule* is a rule that is of the form $A \to B$, where A and B are variables. In the third step, we eliminate all unit-rules from G_2.

We consider all unit-rules, one after another. Let $A \to B$ be one such rule, where A and B are elements of V_2. We know that $B \neq S_2$. We modify G_2 as follows:

1. Remove the rule $A \to B$ from the current set R_2.

2. For each rule in the current set R_2 that is of the form $B \to u$, where $u \in (V_2 \cup \Sigma)^*$, add the rule $A \to u$ to the current set R_2, unless this is a unit-rule that has already been eliminated.

   Observe that in this way, we replace the two-step derivation $A \Rightarrow B \Rightarrow u$ by the one-step derivation $A \Rightarrow u$.

We repeat this process until all unit-rules have been eliminated. Let R_3 be the set of rules, after all unit-rules have been eliminated. We define $G_3 = (V_3, \Sigma, R_3, S_3)$, where $V_3 = V_2$ and $S_3 = S_2$. This grammar has the property that

- the start variable S_3 does not occur on the right-hand side of any rule in R_3,
- R_3 does not contain any ϵ-rule (it may contain the rule $S_3 \to \epsilon$),
- R_3 does not contain any unit-rule, and
- $L(G_3) = L(G_2) = L(G_1) = L(G)$.

Step 4: Eliminate all rules having more than two symbols on the right-hand side.

For each rule in the current set R_3 that is of the form $A \to u_1 u_2 \ldots u_k$, where $k \geq 3$ and each u_i is an element of $V_3 \cup \Sigma$, we modify G_3 as follows:

1. Remove the rule $A \to u_1 u_2 \ldots u_k$ from the current set R_3.

2. Add the following rules to the current set R_3:

$$
\begin{array}{lcl}
A & \to & u_1 A_1 \\
A_1 & \to & u_2 A_2 \\
A_2 & \to & u_3 A_3 \\
 & \vdots & \\
A_{k-3} & \to & u_{k-2} A_{k-2} \\
A_{k-2} & \to & u_{k-1} u_k
\end{array}
$$

where $A_1, A_2, \ldots, A_{k-2}$ are new variables that are added to the current set V_3.

Observe that in this way, we replace the one-step derivation $A \Rightarrow u_1u_2 \ldots u_k$ by the $(k-1)$-step derivation

$$A \Rightarrow u_1A_1 \Rightarrow u_1u_2A_2 \Rightarrow \ldots \Rightarrow u_1u_2 \ldots u_{k-2}A_{k-2} \Rightarrow u_1u_2 \ldots u_k.$$

Let R_4 be the set of rules, and let V_4 be the set of variables, after all rules with more than two symbols on the right-hand side have been eliminated. We define $G_4 = (V_4, \Sigma, R_4, S_4)$, where $S_4 = S_3$. This grammar has the property that

- the start variable S_4 does not occur on the right-hand side of any rule in R_4,
- R_4 does not contain any ϵ-rule (it may contain the rule $S_4 \to \epsilon$),
- R_4 does not contain any unit-rule,
- R_4 does not contain any rule with more than two symbols on the right-hand side, and
- $L(G_4) = L(G_3) = L(G_2) = L(G_1) = L(G)$.

Step 5: Eliminate all rules of the form $A \to u_1u_2$, where u_1 and u_2 are not both variables.

For each rule in the current set R_4 that is of the form $A \to u_1u_2$, where u_1 and u_2 are elements of $V_4 \cup \Sigma$, but u_1 and u_2 are not both contained in V_4, we modify G_3 as follows:

1. If $u_1 \in \Sigma$ and $u_2 \in V_4$, then replace the rule $A \to u_1u_2$ in the current set R_4 by the two rules $A \to U_1u_2$ and $U_1 \to u_1$, where U_1 is a new variable that is added to the current set V_4.

   Observe that in this way, we replace the one-step derivation $A \Rightarrow u_1u_2$ by the two-step derivation $A \Rightarrow U_1u_2 \Rightarrow u_1u_2$.

2. If $u_1 \in V_4$ and $u_2 \in \Sigma$, then replace the rule $A \to u_1u_2$ in the current set R_4 by the two rules $A \to u_1U_2$ and $U_2 \to u_2$, where U_2 is a new variable that is added to the current set V_4.

   Observe that in this way, we replace the one-step derivation $A \Rightarrow u_1u_2$ by the two-step derivation $A \Rightarrow u_1U_2 \Rightarrow u_1u_2$.

3. If $u_1 \in \Sigma$, $u_2 \in \Sigma$, and $u_1 \neq u_2$, then replace the rule $A \to u_1u_2$ in the current set R_4 by the three rules $A \to U_1U_2$, $U_1 \to u_1$, and $U_2 \to u_2$, where U_1 and U_2 are new variables that are added to the current set V_4.

   Observe that in this way, we replace the one-step derivation $A \Rightarrow u_1u_2$ by the three-step derivation $A \Rightarrow U_1U_2 \Rightarrow u_1U_2 \Rightarrow u_1u_2$.

4. If $u_1 \in \Sigma$, $u_2 \in \Sigma$, and $u_1 = u_2$, then replace the rule $A \to u_1u_2 = u_1u_1$ in the current set R_4 by the two rules $A \to U_1U_1$ and $U_1 \to u_1$, where U_1 is a new variable that is added to the current set V_4.

   Observe that in this way, we replace the one-step derivation $A \Rightarrow u_1u_2 = u_1u_1$ by the three-step derivation $A \Rightarrow U_1U_1 \Rightarrow u_1U_1 \Rightarrow u_1u_1$.

Let R_5 be the set of rules, and let V_5 be the set of variables, after Step 5 has been completed. We define $G_5 = (V_5, \Sigma, R_5, S_5)$, where $S_5 = S_4$. This grammar has the property that

- the start variable S_5 does not occur on the right-hand side of any rule in R_5,
- R_5 does not contain any ϵ-rule (it may contain the rule $S_5 \to \epsilon$),
- R_5 does not contain any unit-rule,
- R_5 does not contain any rule with more than two symbols on the right-hand side,
- R_5 does not contain any rule of the form $A \to u_1u_2$, where u_1 and u_2 are not both variables of V_5, and
- $L(G_5) = L(G_4) = L(G_3) = L(G_2) = L(G_1) = L(G)$.

Since the grammar G_5 is in Chomsky normal form, the proof is complete. ∎

3.4.1 An example

Consider the context-free grammar $G = (V, \Sigma, R, A)$, where $V = \{A, B\}$, $\Sigma = \{0, 1\}$, A is the start variable, and R consists of the rules

$$\begin{array}{lcl} A & \rightarrow & BAB|B|\epsilon \\ B & \rightarrow & 00|\epsilon \end{array}$$

We apply the construction given in the proof of Theorem 3.4.2 to convert this grammar to a context-free grammar in Chomsky normal form whose language is the same as that of G. Throughout the construction, upper case letters will denote variables.

Step 1: Eliminate the start variable from the right-hand side of the rules.

We introduce a new start variable S, and add the rule $S \rightarrow A$. This gives the following grammar:

$$\begin{array}{lcl} S & \rightarrow & A \\ A & \rightarrow & BAB|B|\epsilon \\ B & \rightarrow & 00|\epsilon \end{array}$$

Step 2: Eliminate all ϵ-rules.

We take the ϵ-rule $A \rightarrow \epsilon$, and remove it. Then we consider all rules that contain A on the right-hand side. There are two such rules:

- $S \rightarrow A$; we add the rule $S \rightarrow \epsilon$;
- $A \rightarrow BAB$; we add the rule $A \rightarrow BB$.

This gives the following grammar:

$$\begin{array}{lcl} S & \rightarrow & A|\epsilon \\ A & \rightarrow & BAB|B|BB \\ B & \rightarrow & 00|\epsilon \end{array}$$

We take the ϵ-rule $B \rightarrow \epsilon$, and remove it. Then we consider all rules that contain B on the right-hand side. There are three such rules:

- $A \rightarrow BAB$; we add the rules $A \rightarrow AB$, $A \rightarrow BA$, and $A \rightarrow A$;
- $A \rightarrow B$; we do not add the rule $A \rightarrow \epsilon$, because it has already been removed;

- $A \to BB$; we add the rule $A \to B$, but not the rule $A \to \epsilon$ (because it has already been removed).

At this moment, we have the following grammar:

$$\begin{array}{lcl} S & \to & A|\epsilon \\ A & \to & BAB|B|BB|AB|BA|A \\ B & \to & 00 \end{array}$$

Since all ϵ-rules have been eliminated, this completes Step 2. (Observe that the rule $S \to \epsilon$ is allowed, because S is the start variable.)

Step 3: Eliminate all unit-rules.

We take the unit-rule $A \to A$. We can remove this rule, without adding any new rule. At this moment, we have the following grammar:

$$\begin{array}{lcl} S & \to & A|\epsilon \\ A & \to & BAB|B|BB|AB|BA \\ B & \to & 00 \end{array}$$

We take the unit-rule $S \to A$, remove it, and add the rules

$$S \to BAB|B|BB|AB|BA.$$

This gives the following grammar:

$$\begin{array}{lcl} S & \to & \epsilon|BAB|B|BB|AB|BA \\ A & \to & BAB|B|BB|AB|BA \\ B & \to & 00 \end{array}$$

We take the unit-rule $S \to B$, remove it, and add the rule $S \to 00$. This gives the following grammar:

$$\begin{array}{lcl} S & \to & \epsilon|BAB|BB|AB|BA|00 \\ A & \to & BAB|B|BB|AB|BA \\ B & \to & 00 \end{array}$$

We take the unit-rule $A \to B$, remove it, and add the rule $A \to 00$. This gives the following grammar:

$$\begin{array}{lcl} S & \to & \epsilon|BAB|BB|AB|BA|00 \\ A & \to & BAB|BB|AB|BA|00 \\ B & \to & 00 \end{array}$$

Since all unit-rules have been eliminated, this concludes Step 3.

Step 4: Eliminate all rules having more than two symbols on the right-hand side. There are two such rules:

- We take the rule $S \to BAB$, remove it, and add the rules $S \to BA_1$ and $A_1 \to AB$.
- We take the rule $A \to BAB$, remove it, and add the rules $A \to BA_2$ and $A_2 \to AB$.

This gives the following grammar:

$$\begin{array}{lcl} S & \to & \epsilon|BB|AB|BA|00|BA_1 \\ A & \to & BB|AB|BA|00|BA_2 \\ B & \to & 00 \\ A_1 & \to & AB \\ A_2 & \to & AB \end{array}$$

Step 4 is now completed.

Step 5: Eliminate all rules, whose right-hand side contains exactly two symbols, which are not both variables. There are three such rules:

- We replace the rule $S \to 00$ by the rules $S \to A_3A_3$ and $A_3 \to 0$.
- We replace the rule $A \to 00$ by the rules $A \to A_4A_4$ and $A_4 \to 0$.
- We replace the rule $B \to 00$ by the rules $B \to A_5A_5$ and $A_5 \to 0$.

This gives the following grammar, which is in Chomsky normal form:

$$\begin{array}{lcl} S & \to & \epsilon|BB|AB|BA|BA_1|A_3A_3 \\ A & \to & BB|AB|BA|BA_2|A_4A_4 \\ B & \to & A_5A_5 \\ A_1 & \to & AB \\ A_2 & \to & AB \\ A_3 & \to & 0 \\ A_4 & \to & 0 \\ A_5 & \to & 0 \end{array}$$

3.5 Pushdown automata

In this section, we introduce nondeterministic pushdown automata. As we will see, the class of languages that can be accepted by these automata is exactly the class of context-free languages.

We start with an informal description of a *deterministic* pushdown automaton. Such an automaton consists of the following, see also Figure 3.1.

1. There is a *tape* which is divided into *cells*. Each cell stores a symbol belonging to a finite set Σ, called the *tape alphabet*. There is a special symbol $\Box$ that is not contained in Σ; this symbol is called the *blank symbol*. If a cell contains $\Box$, then this means that the cell is actually empty.

2. There is a *tape head* which can move along the tape, one cell to the right per move. This tape head can also read the cell it currently scans.

3. There is a *stack* containing symbols from a finite set Γ, called the *stack alphabet*. This set contains a special symbol \$.

4. There is a *stack head* which can read the top symbol of the stack. This head can also *pop* the top symbol, and it can *push* symbols of Γ onto the stack.

5. There is a *state control*, which can be in any one of a finite number of *states*. The set of states is denoted by Q. The set Q contains one special state q, called the *start state*.

The input for a pushdown automaton is a string in Σ^*. This input string is stored on the tape of the pushdown automaton and, initially, the tape head is on the leftmost symbol of the input string. Initially, the stack only contains the special symbol \$, and the pushdown automaton is in the start state q. In one computation step, the pushdown automaton does the following:

1. Assume that the pushdown automaton is currently in state r. Let a be the symbol of Σ that is read by the tape head, and let A be the symbol of Γ that is on top of the stack.

2. Depending on the current state r, the tape symbol a, and the stack symbol A,

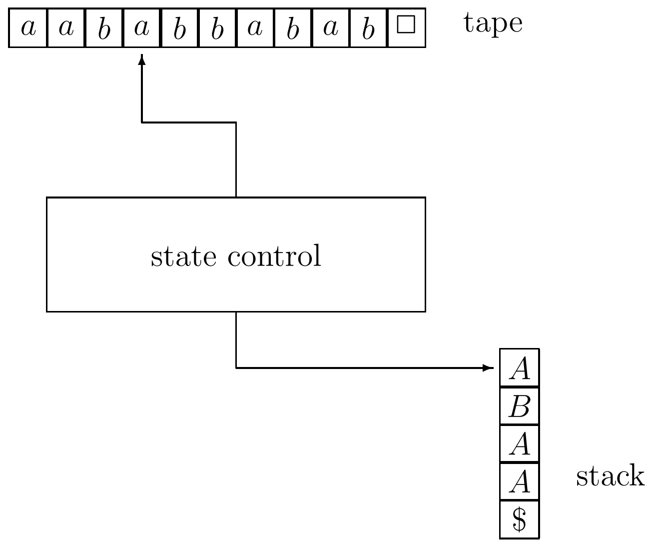


Figure 3.1: *A pushdown automaton.*

(a) the pushdown automaton switches to a state r' of Q (which may be equal to r),

(b) the tape head either moves one cell to the right or stays at the current cell, and

(c) the top symbol A is replaced by a string w that belongs to Γ^*. To be more precise,

   i. if $w = \epsilon$, then A is popped from the stack, whereas

   ii. if $w = B_1 B_2 \dots B_k$, with $k \geq 1$ and $B_1, B_2, \dots, B_k \in \Gamma$, then A is replaced by w, and B_k becomes the new top symbol of the stack.

Later, we will specify when the pushdown automaton accepts the input string.

We now give a formal definition of a deterministic pushdown automaton.

Definition 3.5.1 A *deterministic pushdown automaton* is a 5-tuple $M = (\Sigma, \Gamma, Q, \delta, q)$, where

1. Σ is a finite set, called the *tape alphabet*; the blank symbol $\Box$ is not contained in Σ,

2. Γ is a finite set, called the *stack alphabet*; this alphabet contains the special symbol \$,

3. Q is a finite set, whose elements are called *states*,

4. q is an element of Q; it is called the *start state*,

5. δ is called the *transition function*, which is a function

$$\delta : Q \times (\Sigma \cup \{\Box\}) \times \Gamma \rightarrow Q \times \{N, R\} \times \Gamma^*.$$

The transition function δ can be thought of as being the "program" of the pushdown automaton. This function tells us what the automaton can do in one "computation step": Let $r \in Q$, $a \in \Sigma \cup \{\Box\}$, and $A \in \Gamma$. Furthermore, let $r' \in Q$, $\sigma \in \{R, N\}$, and $w \in \Gamma^*$ be such that

$$\delta(r, a, A) = (r', \sigma, w). \tag{3.1}$$

This transition means that if

- the pushdown automaton is in state r,
- the tape head reads the symbol a, and
- the top symbol on the stack is A,

then

- the pushdown automaton switches to state r',
- the tape head moves according to σ: if $\sigma = R$, then it moves one cell to the right; if $\sigma = N$, then it does not move, and
- the top symbol A on the stack is replaced by the string w.

We will write the computation step (3.1) in the form of the *instruction*

$$raA \rightarrow r'\sigma w.$$

We now specify the computation of the pushdown automaton $M = (\Sigma, \Gamma, Q, \delta, q)$.

Start configuration: Initially, the pushdown automaton is in the start state q, the tape head is on the leftmost symbol of the input string $a_1a_2\dots a_n$, and the stack only contains the special symbol \$.

Computation and termination: Starting in the start configuration, the pushdown automaton performs a sequence of computation steps as described above. It *terminates* at the moment when the stack becomes empty. (Hence, if the stack never gets empty, the pushdown automaton does *not* terminate.)

Acceptance: The pushdown automaton *accepts* the input string $a_1a_2\dots a_n \in \Sigma^*$, if

1. the automaton terminates on this input, and
2. at the time of termination (i.e., at the moment when the stack gets empty), the tape head is on the cell immediately to the right of the cell containing the symbol a_n (this cell must contain the blank symbol $\Box$).

In all other cases, the pushdown automaton *rejects* the input string. Thus, the pushdown automaton rejects this string if

1. the automaton does not terminate on this input (i.e., the computation "loops forever") or
2. at the time of termination, the tape head is not on the cell immediately to the right of the cell containing the symbol a_n.

We denote by $L(M)$ the language *accepted* by the pushdown automaton M. Thus,

$$L(M) = \{w \in \Sigma^* : \ M \text{ accepts } w\}.$$

The pushdown automaton described above is deterministic. For a *nondeterministic* pushdown automata, the current computation step may not be uniquely defined, but the automaton can make a choice out of a finite number of possibilities. In this case, the transition function δ is a function

$$\delta : Q \times (\Sigma \cup \{\Box\}) \times \Gamma \rightarrow \mathcal{P}_f(Q \times \{N, R\} \times \Gamma^*),$$

where $\mathcal{P}_f(K)$ is the set of all *finite* subsets of the set K.

We say that a nondeterministic pushdown automaton M *accepts* an input string, if there *exists* an accepting computation, in the sense as described for deterministic pushdown automata. We say that M *rejects* an input string, if *every* computation on this string is rejecting. As before, we denote by $L(M)$ the set of all strings in Σ^* that are accepted by M.

3.6 Examples of pushdown automata

3.6.1 Properly nested parentheses

We will show how to construct a deterministic pushdown automaton, that accepts the set of all strings of properly nested parentheses. Observe that a string w in $\{(,)\}^*$ is properly nested if and only if

- in every prefix of w, the number of "(" is greater than or equal to the number of ")", and
- in the complete string w, the number of "(" is equal to the number of ")".

We will use the tape symbol a for "(", and the tape symbol b for ")".

The idea is as follows. Recall that initially, the stack only contains the special symbol \$. The pushdown automaton reads the input string from left to right. For every a it reads, it pushes the symbol S onto the stack, and for every b it reads, it pops the top symbol from the stack. In this way, the number of symbols S on the stack will always be equal to the number of as that have been read minus the number of bs that have been read; additionally, the bottom of the stack will contain the special symbol \$. The input string is properly nested if and only if (i) this difference is always non-negative and (ii) this difference is zero once the entire input string has been read. Hence, the input string is accepted if and only if during this process, (i) the stack always contains at least the special symbol \$ and (ii) at the end, the stack only contains the special symbol \$ (which will then be popped in the final step).

Based on this discussion, we obtain the deterministic pushdown automaton $M = (\Sigma, \Gamma, Q, \delta, q)$, where $\Sigma = \{a, b\}$, $\Gamma = \{\$, S\}$, $Q = \{q\}$, and the transition function δ is specified by the following instructions:

$qa\$ \to qR\$S$	because of the a, S is pushed onto the stack
$qaS \to qRSS$	because of the a, S is pushed onto the stack
$qbS \to qR\epsilon$	because of the b, the top element is popped from the stack
$qb\$ \to qN\epsilon$	the number of bs read is larger than the number of as read; the stack is made empty (hence, the computation terminates before the entire string has been read), and the input string is rejected
$q\square\$ \to qN\epsilon$	the entire input string has been read; the stack is made empty, and the input string is accepted
$q\square S \to qNS$	the entire input string has been read, it contains more as than bs; no changes are made (thus, the automaton does not terminate), and the input string is rejected

3.6.2 Strings of the form 0^n1^n

We construct a deterministic pushdown automata that accepts the language $\{0^n1^n : n \geq 0\}$.

The automaton uses two states q_0 and q_1, where q_0 is the start state. Initially, the automaton is in state q_0.

- For each 0 that it reads, the automaton pushes one symbol S onto the stack and stays in state q_0.

- When the first 1 is read, the automaton switches to state q_1. From that moment,

  - for each 1 that is read, the automaton pops the top symbol from the stack and stays in state q_1;

  - if a 0 is read, the automaton does not make any change and, therefore, does not terminate.

Based on this discussion, we obtain the deterministic pushdown automaton $M = (\Sigma, \Gamma, Q, \delta, q_0)$, where $\Sigma = \{0, 1\}$, $\Gamma = \{\$, S\}$, $Q = \{q_0, q_1\}$, q_0 is the start state, and the transition function δ is specified by the following instructions:

$q_0 0\$ \rightarrow q_0 R\$S$	push S onto the stack
$q_0 0S \rightarrow q_0 RSS$	push S onto the stack
$q_0 1\$ \rightarrow q_0 N\$$	first symbol in the input is 1; loop forever
$q_0 1S \rightarrow q_1 R\epsilon$	first 1 is encountered
$q_0 \square\$ \rightarrow q_0 N\epsilon$	input string is empty; accept
$q_0 \square S \rightarrow q_0 NS$	input only consists of 0s; loop forever
$q_1 0\$ \rightarrow q_1 N\$$	0 to the right of 1; loop forever
$q_1 0S \rightarrow q_1 NS$	0 to the right of 1; loop forever
$q_1 1\$ \rightarrow q_1 N\$$	too many 1s; loop forever
$q_1 1S \rightarrow q_1 R\epsilon$	pop top symbol from the stack
$q_1 \square\$ \rightarrow q_1 N\epsilon$	accept
$q_1 \square S \rightarrow q_1 NS$	too many 0s; loop forever

3.6.3 Strings with b in the middle

We will construct a nondeterministic pushdown automaton that accepts the set L of all strings in $\{a, b\}^*$ having an odd length and whose middle symbol is b, i.e.,

$$L = \{vbw : v \in \{a, b\}^*, w \in \{a, b\}^*, |v| = |w|\}.$$

The idea is as follows. The automaton uses two states q and q', where q is the start state. These states have the following meaning:

- If the automaton is in state q, then it has not reached the middle symbol b of the input string.
- If the automaton is in state q', then it has read the middle symbol b.

Observe that since the automaton can only make one single pass over the input string, it has to "guess" (i.e., use nondeterminism) when it reaches the middle of the string.

- If the automaton is in state q, then, when reading the current tape symbol,
  - it either pushes one symbol S onto the stack and stays in state q
  - or, in case the current tape symbol is b, it "guesses" that it has reached the middle of the input string, by switching to state q'.
- If the automaton is in state q', then, when reading the current tape symbol, it pops the top symbol S from the stack and stays in state q'.

In this way, the number of symbols S on the stack will always be equal to the difference of (i) the number of symbols in the part to the left of the middle symbol b that have been read and (ii) the number of symbols in the part to the right of the middle symbol b that have been read; additionally, the bottom of the stack will contain the special symbol \$.

The input string is accepted if and only if, at the moment when the blank symbol $\Box$ is read, the automaton is in state q' and the top symbol on the stack is \$. In this case, the stack is made empty and, thus, the computation terminates.

We obtain the nondeterministic pushdown automaton $M = (\Sigma, \Gamma, Q, \delta, q)$, where $\Sigma = \{a, b\}$, $\Gamma = \{\$, S\}$, $Q = \{q, q'\}$, q is the start state, and the transition function δ is specified by the following instructions:

$qa\$ \to qR\$S$	push S onto the stack
$qaS \to qRSS$	push S onto the stack
$qb\$ \to q'R\$$	reached the middle
$qb\$ \to qR\$S$	did not reach the middle; push S onto the stack
$qbS \to q'RS$	reached the middle
$qbS \to qRSS$	did not reach the middle; push S onto the stack
$q\Box\$ \to qN\$$	input string is empty; loop forever
$q\Box S \to qNS$	loop forever
$q'a\$ \to q'N\epsilon$	stack is empty; terminate, but reject, because the entire input string has not been read
$q'aS \to q'R\epsilon$	pop top symbol from stack
$q'b\$ \to q'N\epsilon$	stack is empty; terminate, but reject, because the entire input string has not been read
$q'bS \to q'R\epsilon$	pop top symbol from stack
$q'\Box\$ \to q'N\epsilon$	accept
$q'\Box S \to q'NS$	loop forever

Remark 3.6.1 It can be shown that there is no deterministic pushdown automaton that accepts the language L. The reason is that a deterministic pushdown automaton cannot determine when it reaches the middle of the input string. Thus, unlike as for finite automata, nondeterministic pushdown automata are *more powerful* than their deterministic counterparts.

3.7 Equivalence of pushdown automata and context-free grammars

The main result of this section is that *nondeterministic* pushdown automata and context-free grammars are equivalent in power:

Theorem 3.7.1 *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. Then A is context-free if and only if there exists a nondeterministic pushdown automaton that accepts A.*

We will only prove one direction of this theorem. That is, we will show how to convert an arbitrary context-free grammar to a nondeterministic pushdown automaton.

Let $G = (V, \Sigma, \mathcal{R}, \$)$ be a context-free grammar, where V is the set of variables, Σ is the set of terminals, $\mathcal{R}$ is the set of rules, and $\$$ is the start variable. By Theorem 3.4.2, we may assume that G is in Chomsky normal form. Hence, every rule in $\mathcal{R}$ has one of the following three forms:

1. $A \to BC$, where A, B, and C are variables, $B \neq \$$, and $C \neq \$$.
2. $A \to a$, where A is a variable and a is a terminal.
3. $\$ \to \epsilon$.

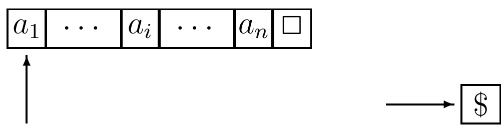
We will construct a nondeterministic pushdown automaton M that accepts the language $L(G)$ of this grammar G. Observe that M must have the following property: For every string $w = a_1 a_2 \ldots a_n \in \Sigma^*$,

$$w \in L(G) \text{ if and only if } M \text{ accepts } w.$$

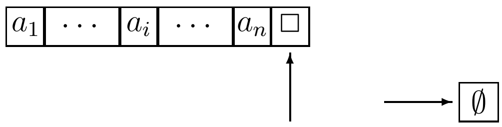
This can be reformulated as follows:

$$\$ \stackrel{*}{\Rightarrow} a_1 a_2 \ldots a_n$$

if and only if there exists a computation of M that starts in the initial configuration



and ends in the configuration



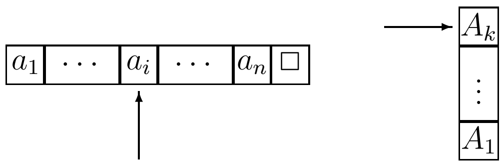
where $\boxed{\emptyset}$ indicates that the stack is empty.

Assume that $\$ \overset{*}{\Rightarrow} a_1a_2\dots a_n$. Then there exists a derivation (using the rules of $\mathcal{R}$) of the string $a_1a_2\dots a_n$ from the start variable \$. We may assume that in each step in this derivation, a rule is applied to the leftmost variable in the current string. Hence, because the grammar G is in Chomsky normal form, at any moment during the derivation, the current string has the form

$$a_1a_2\dots a_{i-1}A_kA_{k-1}\dots A_1, \tag{3.2}$$

for some integers i and k with $1 \leq i \leq n+1$ and $k \geq 0$, and variables $A_1, A_2, \dots, A_k$. (In particular, at the start of the derivation, we have $i = 1$ and $k = 1$, and the current string is $A_k = \$$. At the end of the derivation, we have $i = n+1$ and $k = 0$, and the current string is $a_1a_2\dots a_n$.)

We will *define* the pushdown automaton M in such a way that the current string (3.2) corresponds to the configuration



Based on this discussion, we obtain the nondeterministic pushdown automaton $M = (\Sigma, V, \{q\}, \delta, q)$, where

- the tape alphabet is the set Σ of terminals of G,
- the stack alphabet is the set V of variables of G,
- the set of states consists of one state q, which is the start state, and
- the transition function δ is obtained from the rules in $\mathcal{R}$, in the following way:

– For each rule in $\mathcal{R}$ that is of the form $A \to BC$, with $A, B, C \in V$, the pushdown automaton M has the instructions

$$qaA \to qNCB, \text{ for all } a \in \Sigma.$$

– For each rule in $\mathcal{R}$ that is of the form $A \to a$, with $A \in V$ and $a \in \Sigma$, the pushdown automaton M has the instruction

$$qaA \to qR\epsilon.$$

– If $\mathcal{R}$ contains the rule $\$ \to \epsilon$, then the pushdown automaton M has the instruction

$$q\Box\$ \to qN\epsilon.$$

This concludes the definition of M. It remains to prove that $L(M) = L(G)$, i.e., the language of the nondeterministic pushdown automaton M is equal to the language of the context-free grammar G. Hence, we have to show that for every string $w \in \Sigma^*$,

$$w \in L(G) \text{ if and only if } w \in L(M),$$

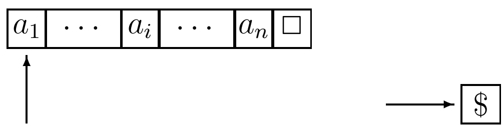
which can be rewritten as

$$\$ \stackrel{*}{\Rightarrow} w \text{ if and only if } M \text{ accepts } w.$$

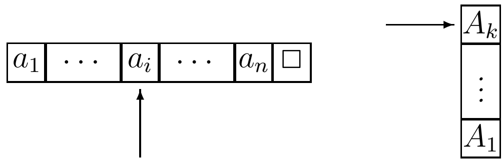
Claim 3.7.2 *Let $a_1a_2 \ldots a_n$ be a string in Σ^*, let $A_1, A_2, \ldots, A_k$ be variables in V, and let i and k be integers with $1 \leq i \leq n+1$ and $k \geq 0$. Then the following holds:*

$$\$ \stackrel{*}{\Rightarrow} a_1a_2 \ldots a_{i-1}A_kA_{k-1} \ldots A_1$$

if and only if there exists a computation of M from the initial configuration



to the configuration



Proof. The claim can be proved by induction. Let

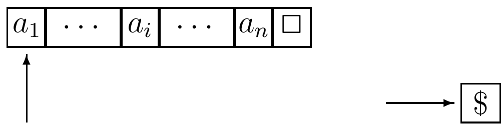
$$w = a_1 a_2 \ldots a_{i-1} A_k A_{k-1} \ldots A_1.$$

Assume that $k \geq 1$ and assume that the claim is true for the string w. Then we have to show that the claim is still true after applying a rule in $\mathcal{R}$ to the leftmost variable A_k in w. Since the grammar is in Chomsky normal form, the rule to be applied is either of the form $A_k \to BC$ or of the form $A_k \to a_i$. In both cases, the property mentioned in the claim is maintained. ∎

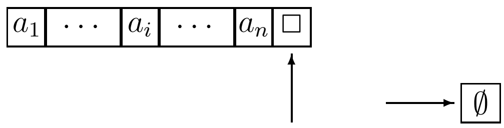
We now use Claim 3.7.2 to prove that $L(M) = L(G)$. Let $w = a_1 a_2 \ldots a_n$ be an arbitrary string in Σ^*. By applying Claim 3.7.2, with $i = n+1$ and $k = 0$, we see that $w \in L(G)$, i.e.,

$$\$ \stackrel{*}{\Rightarrow} a_1 a_2 \ldots a_n,$$

if and only if there exists a computation of M from the initial configuration



to the configuration



But this means that $w \in L(G)$ if and only if the automaton M accepts the string w.

This concludes the proof of the fact that every context-free grammar can be converted to a nondeterministic pushdown automaton. As mentioned already, we will not give the conversion in the other direction. We finish this section with the following observation:

Theorem 3.7.3 *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a context-free language. Then there exists a nondeterministic pushdown automaton that accepts A and has only one state.*

Proof. Since A is context-free, there exists a context-free grammar G_0 such that $L(G_0) = A$. By Theorem 3.4.2, there exists a context-free grammar G that is in Chomsky normal form and for which $L(G) = L(G_0)$. The construction given above converts G to a nondeterministic pushdown automaton M that has only one state and for which $L(M) = L(G)$. ∎

3.8 The pumping lemma for context-free languages

In Section 2.9, we proved the pumping lemma for regular languages and used it to prove that certain languages are not regular. In this section, we generalize the pumping lemma to context-free languages. The idea is to consider the *parse tree* (see Section 3.1) that describes the derivation of a sufficiently long string in the context-free language L. Since the number of variables in the corresponding context-free grammar G is finite, there is at least one variable, say A_j, that occurs more than once on the longest root-to-leaf path in the parse tree. The subtree which is sandwiched between two occurrences of A_j on this path can be copied any number of times. This will result in a legal parse tree and, hence, in a "pumped" string that is in the language L.

Theorem 3.8.1 (Pumping Lemma for Context-Free Languages) *Let L be a context-free language. Then there exists an integer $p \geq 1$, called the* pumping length, *such that the following holds: Every string s in L, with $|s| \geq p$, can be written as $s = uvxyz$, such that*

1. *$|vy| \geq 1$ (i.e., v and y are not both empty),*
2. *$|vxy| \leq p$, and*
3. *$uv^ixy^iz \in L$, for all $i \geq 0$.*

3.8.1 Proof of the pumping lemma

The proof of the pumping lemma will use the following result about parse trees:

Lemma 3.8.2 *Let G be a context-free grammar in Chomsky normal form, let s be a non-empty string in $L(G)$, and let T be a parse tree for s. Let ℓ be the height of T, i.e., ℓ is the number of edges on a longest root-to-leaf path in T. Then*

$$|s| \leq 2^{\ell-1}.$$

Proof. The claim can be proved by induction on ℓ. By looking at some small values of ℓ and using the fact that G is in Chomsky normal form, you should be able to verify the claim. ∎

Now we can start with the proof of the pumping lemma. Let L be a context-free language and let Σ be the alphabet of L. By Theorem 3.4.2, there exists a context-free grammar in Chomsky normal form, $G = (V, \Sigma, R, S)$, such that $L = L(G)$.

Define r to be the number of variables of G and define $p = 2^r$. We will prove that the value of p can be used as the pumping length. Consider an arbitrary string s in L such that $|s| \geq p$, and let T be a parse tree for s. Let ℓ be the height of T. Then, by Lemma 3.8.2, we have

$$|s| \leq 2^{\ell-1}.$$

On the other hand, we have

$$|s| \geq p = 2^r.$$

By combining these inequalities, we see that $2^r \leq 2^{\ell-1}$, which can be rewritten as

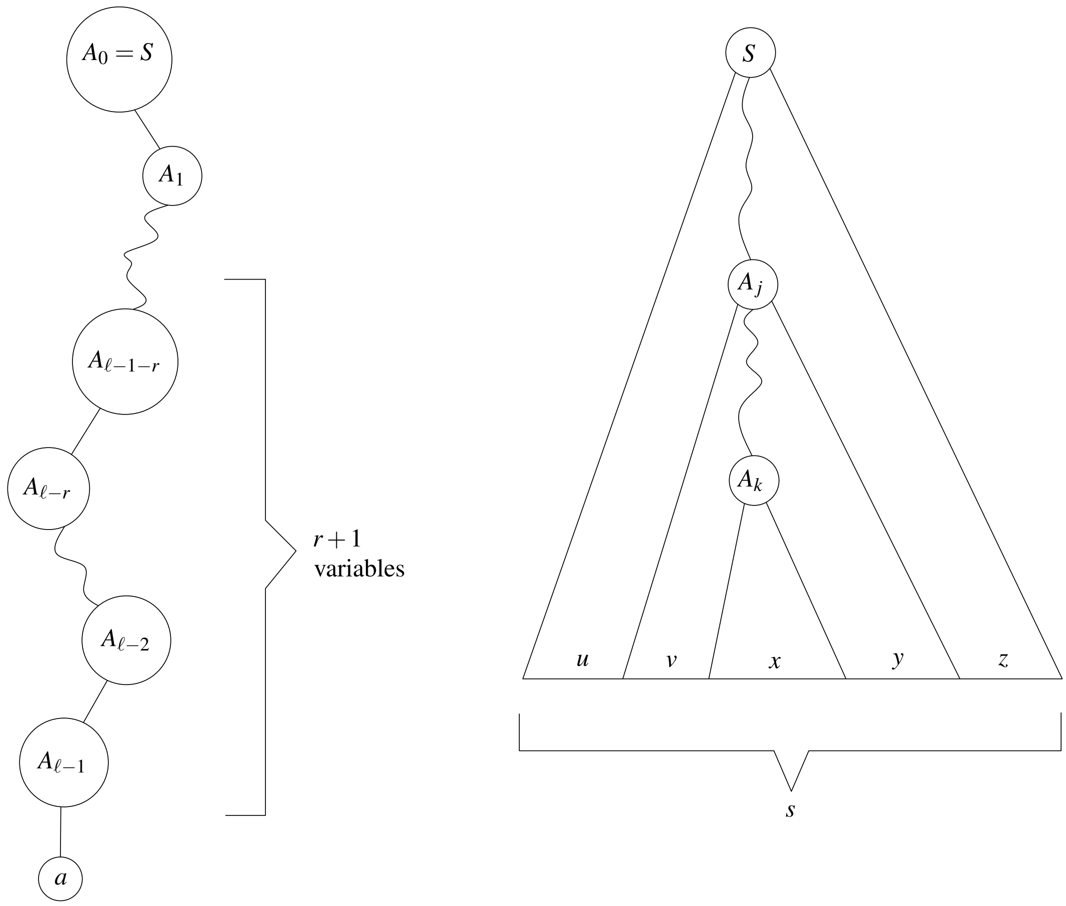
$$\ell \geq r + 1.$$

Consider the nodes on a longest root-to-leaf path in T. Since this path consists of ℓ edges, it consists of $\ell + 1$ nodes. The first ℓ of these nodes store variables, which we denote by $A_0, A_1, \ldots, A_{\ell-1}$ (where $A_0 = S$), and the last node (which is a leaf) stores a terminal, which we denote by a.

Since $\ell - 1 - r \geq 0$, the sequence

$$A_{\ell-1-r}, A_{\ell-r}, \ldots, A_{\ell-1}$$

of variables is well-defined. Observe that this sequence consists of $r + 1$ variables. Since the number of variables in the grammar G is equal to r, the pigeonhole principle implies that there is a variable that occurs at least twice in this sequence. In other words, there are indices j and k, such that $\ell - 1 - r \leq j < k \leq \ell - 1$ and $A_j = A_k$. Refer to the figure below for an illustration.



Recall that T is a parse tree for the string s. Therefore, the terminals stored at the leaves of T, in the order from left to right, form s. As indicated in the figure above, the nodes storing the variables A_j and A_k partition s into five substrings u, v, x, y, and z, such that $s = uvxyz$.

It remains to prove that the three properties stated in the pumping lemma hold. We start with the third property, i.e., we prove that

$$uv^ixy^iz \in L, \text{ for all } i \geq 0.$$

In the grammar G, we have

$$S \overset{*}{\Rightarrow} uA_jz. \tag{3.3}$$

Since $A_j \overset{*}{\Rightarrow} vA_ky$ and $A_k = A_j$, we have

$$A_j \overset{*}{\Rightarrow} vA_jy. \tag{3.4}$$

Finally, since $A_k \overset{*}{\Rightarrow} x$ and $A_k = A_j$, we have

$$A_j \overset{*}{\Rightarrow} x. \tag{3.5}$$

From (3.3) and (3.5), it follows that

$$S \overset{*}{\Rightarrow} uA_jz \overset{*}{\Rightarrow} uxz,$$

which implies that the string uxz is in the language L. Similarly, it follows from (3.3), (3.4), and (3.5) that

$$S \overset{*}{\Rightarrow} uA_jz \overset{*}{\Rightarrow} uvA_jyz \overset{*}{\Rightarrow} uvvA_jyyz \overset{*}{\Rightarrow} uvvxyyz.$$

Hence, the string uv^2xy^2z is in the language L. In general, for each $i \geq 0$, the string uv^ixy^iz is in the language L, because

$$S \overset{*}{\Rightarrow} uA_jz \overset{*}{\Rightarrow} uv^iA_jy^iz \overset{*}{\Rightarrow} uv^ixy^iz.$$

This proves that the third property in the pumping lemma holds.

Next we show that the second property holds. That is, we prove that $|vxy| \leq p$. Consider the subtree rooted at the node storing the variable A_j. The path from the node storing A_j to the leaf storing the terminal a is a longest path in this subtree. (Convince yourself that this is true.) Moreover, this path consists of $\ell - j$ edges. Since $A_j \overset{*}{\Rightarrow} vxy$, this subtree is a parse tree for the string vxy (where A_j is used as the start variable). Therefore, by Lemma 3.8.2, we can conclude that $|vxy| \leq 2^{\ell-j-1}$. We know that $\ell - 1 - r \leq j$, which is equivalent to $\ell - j - 1 \leq r$. It follows that

$$|vxy| \leq 2^{\ell-j-1} \leq 2^r = p.$$

Finally, we show that the first property in the pumping lemma holds. That is, we prove that $|vy| \geq 1$. Recall that

$$A_j \overset{*}{\Rightarrow} vA_k y.$$

Let the first rule used in this derivation be $A_j \to BC$. (Since the variables A_j and A_k, even though they are equal, are stored at different nodes of the parse tree, and since the grammar G is in Chomsky normal form, this first rule exists.) Then

$$A_j \Rightarrow BC \overset{*}{\Rightarrow} vA_k y.$$

Observe that the string BC has length two. Moreover, by applying rules of a grammar in Chomsky normal form, strings cannot become shorter. (Here, we use the fact that the start variable does not occur on the right-hand side of any rule.) Therefore, we have $|vA_k y| \geq 2$. But this implies that $|vy| \geq 1$. This completes the proof of the pumping lemma.

3.8.2 Applications of the pumping lemma

First example

Consider the language

$$A = \{a^n b^n c^n : n \geq 0\}.$$

We will prove by contradiction that A is not a context-free language.

Assume that A is a context-free language. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. Consider the string $s = a^p b^p c^p$. Observe that $s \in A$ and $|s| = 3p \geq p$. Hence, by the pumping lemma, s can be written as $s = uvxyz$, where $|vy| \geq 1$, $|vxy| \leq p$, and $uv^i xy^i z \in A$ for all $i \geq 0$.

Observe that the pumping lemma does not tell us the location of the substring vxy in the string s, it only gives us an upper bound on the length of this substring. Therefore, we have to consider three cases, depending on the location of vxy in s.

Case 1: The substring vxy does not contain any c.

Consider the string $uv^2xy^2z = uvvxyyz$. Since $|vy| \geq 1$, this string contains more than p many as or more than p many bs. Since it contains exactly p many cs, it follows that this string is not in the language A. This is a contradiction because, by the pumping lemma, the string uv^2xy^2z is in A.

Case 2: The substring vxy does not contain any a.

Consider the string $uv^2xy^2z = uvvxyyz$. Since $|vy| \geq 1$, this string contains more than p many bs or more than p many cs. Since it contains exactly p many as, it follows that this string is not in the language A. This is a contradiction because, by the pumping lemma, the string uv^2xy^2z is in A.

Case 3: The substring vxy contains at least one a and at least one c.

Since $s = a^p b^p c^p$, this implies that $|vxy| > p$, which again contradicts the pumping lemma.

Thus, in all of the three cases, we have obtained a contradiction. Therefore, we have shown that the language A is not context-free.

Second example

Consider the languages

$$A = \{ww^R : w \in \{a, b\}^*\},$$

where w^R is the string obtained by writing w backwards, and

$$B = \{ww : w \in \{a, b\}^*\}.$$

Even though these languages look similar, we will show that A is context-free and B is not context-free.

Consider the following context-free grammar, in which S is the start variable:

$$S \to \epsilon | aSa | bSb.$$

It is easy to see that the language of this grammar is exactly the language A. Therefore, A is context-free. Alternatively, we can show that A is context-free, by constructing a (nondeterministic) pushdown automaton that accepts A. This automaton has two states q and q', where q is the start state. If the automaton is in state q, then it did not yet finish reading the leftmost half of the input string; it pushes all symbols read onto the stack. If the automaton is in state q', then it is reading the rightmost half of the input string; for each symbol read, it checks whether it is equal to the symbol on top of the stack and, if so, pops the top symbol from the stack. The pushdown automaton uses nondeterminism to "guess" when to switch from state q to state q' (i.e., when it has completed reading the leftmost half of the input string).

At this point, you should convince yourself that the two approaches above, which showed that A is context-free, do *not* work for B. The reason why they do not work is that the language B is not context-free, as we will prove now.

Assume that B is a context-free language. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. At this point, we must choose a string s in B, whose length is at least p, and that does not satisfy the three properties stated in the pumping lemma. Let us try the string $s = a^p b a^p b$. Then $s \in B$ and $|s| = 2p + 2 \geq p$. Hence, by the pumping lemma, s can be written as $s = uvxyz$, where (i) $|vy| \geq 1$, (ii) $|vxy| \leq p$, and (iii) $uv^ixy^iz \in B$ for all $i \geq 0$. It may happen that $p \geq 3$, $u = a^{p-1}$, $v = a$, $x = b$, $y = a$, and $z = a^{p-1}b$. If this is the case, then properties (i), (ii), and (iii) hold, and, thus, we do not get a contradiction. In other words, we have chosen the "wrong" string s. This string is "wrong", because there is only one b between the as. Because of this, v can be in the leftmost block of as, and y can be in the rightmost block of as. Observe that if there were at least p many bs between the as, then this would not happen, because $|vxy| \leq p$.

Based on the discussion above, we choose $s = a^p b^p a^p b^p$. Observe that $s \in B$ and $|s| = 4p \geq p$. Hence, by the pumping lemma, s can be written as $s = uvxyz$, where $|vy| \geq 1$, $|vxy| \leq p$, and $uv^ixy^iz \in B$ for all $i \geq 0$. Based on the location of vxy in the string s, we distinguish three cases:

Case 1: The substring vxy overlaps both the leftmost half and the rightmost half of s.

Since $|vxy| \leq p$, the substring vxy is contained in the "middle" part of s, i.e., vxy is contained in the block $b^p a^p$. Consider the string $uv^0xy^0z = uxz$. Since $|vy| \geq 1$, we know that at least one of v and y is non-empty.

- If $v \neq \epsilon$, then v contains at least one b from the leftmost block of bs in s, whereas y does not contain any b from the rightmost block of bs in s. Therefore, in the string uxz, the leftmost block of bs contains fewer bs than the rightmost block of bs. Hence, the string uxz is not contained in B.

- If $y \neq \epsilon$, then y contains at least one a from the rightmost block of as in s, whereas v does not contain any a from the leftmost block of as in s. Therefore, in the string uxz, the leftmost block of as contains more as than the rightmost block of as. Hence, the string uxz is not contained in B.

In both cases, we conclude that the string uxz is not an element of the language B. But, by the pumping lemma, this string is contained in B.

Case 2: The substring vxy is in the leftmost half of s.

In this case, none of the strings uxz, uv^2xy^2z, uv^3xy^3z, uv^4xy^4z, etc., is contained in B. But, by the pumping lemma, each of these strings is contained in B.

Case 3: The substring vxy is in the rightmost half of s.

This case is symmetric to Case 2: None of the strings uxz, uv^2xy^2z, uv^3xy^3z, uv^4xy^4z, etc., is contained in B. But, by the pumping lemma, each of these strings is contained in B.

To summarize, in each of the three cases, we have obtained a contradiction. Therefore, the language B is not context-free.

Third example

We have seen in Section 3.2.4 that the language

$$\{a^m b^n c^{m+n} : m \geq 0, n \geq 0\}$$

is context-free. Using the pumping lemma for regular languages, it is easy to prove that this language is not regular. In other words, context-free grammars can verify addition, whereas finite automata are not powerful enough for this. We now consider the problem of verifying multiplication: Let A be the language defined as

$$A = \{a^m b^n c^{mn} : m \geq 0, n \geq 0\}.$$

We will prove by contradiction that A is not a context-free language.

Assume that A is context-free. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. Consider the string $s = a^p b^p c^{p^2}$. Then, $s \in A$ and $|s| = 2p + p^2 \geq p$. Hence, by the pumping lemma, s can be written as $s = uvxyz$, where $|vy| \geq 1$, $|vxy| \leq p$, and $uv^ixy^iz \in A$ for all $i \geq 0$.

There are three possible cases, depending on the locations of v and y in the string s.

Case 1: The substring v does not contain any a and does not contain any b, and the substring y does not contain any a and does not contain any b.

Consider the string uv^2xy^2z. Since $|vy| \geq 1$, this string consists of p many as, p many bs, but more than p^2 many cs. Therefore, this string is not contained in A. But, by the pumping lemma, it is contained in A.

Case 2: The substring v does not contain any c and the substring y does not contain any c.

Consider again the string uv^2xy^2z. This string consists of p^2 many cs. Since $|vy| \geq 1$, in this string,

- the number of as is at least $p+1$ and the number of bs is at least p, or
- the number of as is at least p and the number of bs is at least $p+1$.

Therefore, the number of as multiplied by the number of bs is at least $p(p+1)$, which is larger than p^2. Therefore, uv^2xy^2z is not contained in A. But, by the pumping lemma, this string is contained in A.

Case 3: The substring v contains at least one b and the substring y contains at least one c.

Since $|vxy| \leq p$, the substring vy does not contain any a. Thus, we can write $vy = b^j c^k$, where $j \geq 1$ and $k \geq 1$. Consider the string uxz. We can write this string as $uxz = a^p b^{p-j} c^{p^2-k}$. Since, by the pumping lemma, this string is contained in A, we have $p(p-j) = p^2 - k$, which implies that $jp = k$. Thus,

$$|vxy| \geq |vy| = j + k = j + jp \geq 1 + p.$$

But, by the pumping lemma, we have $|vxy| \leq p$.

Observe that, since $|vxy| \leq p$, the above three cases cover all possibilities for the locations of v and y in the string s. In each of the three cases, we have obtained a contradiction. Therefore, the language A is not context-free.

Exercises

3.1 Construct context-free grammars that generate the following languages. In all cases, $\Sigma = \{0, 1\}$.

- $\{0^{2n}1^n : n \geq 0\}$
- $\{w :$ w contains at least three 1s$\}$
- $\{w :$ the length of w is odd and its middle symbol is 0$\}$

- $\{w : w \text{ is a palindrome}\}$.
  A *palindrome* is a string w having the property that $w = w^R$, i.e., reading w from left to right gives the same result as reading w from right to left.

- $\{w : w \text{ starts and ends with the same symbol}\}$

- $\{w : w \text{ starts and ends with different symbols}\}$

3.2 Let $G = (V, \Sigma, R, S)$ be the context-free grammar, where $V = \{A, B, S\}$, $\Sigma = \{0, 1\}$, S is the start variable, and R consists of the rules

$$\begin{array}{lcl} S & \rightarrow & 0S|1A|\epsilon \\ A & \rightarrow & 0B|1S \\ B & \rightarrow & 0A|1B \end{array}$$

Define the following language L:

$$\begin{array}{ll} L = \{w \in \{0,1\}^* : & w \text{ is the binary representation of a non-negative} \\ & \text{integer that is divisible by three } \} \cup \{\epsilon\}. \end{array}$$

Prove that $L = L(G)$. (*Hint:* The variables S, A, and B are used to remember the remainder after division by three.)

3.3 Let $G = (V, \Sigma, R, S)$ be the context-free grammar, where $V = \{A, B, S\}$, $\Sigma = \{a, b\}$, S is the start variable, and R consists of the rules

$$\begin{array}{lcl} S & \rightarrow & aB|bA \\ A & \rightarrow & a|aS|BAA \\ B & \rightarrow & b|bS|ABB \end{array}$$

- Prove that $ababba \in L(G)$.

- Prove that $L(G)$ is the set of all non-empty strings w over the alphabet $\{a, b\}$ such that the number of as in w is equal to the number of bs in w.

3.4 Let A and B be context-free languages over the same alphabet Σ.

- Prove that the union $A \cup B$ of A and B is also context-free.

- Prove that the concatenation AB of A and B is also context-free.

- Prove that the star A^* of A is also context-free.

3.5 Define the following two languages A and B:

$$A = \{a^m b^n c^n : m \geq 0, n \geq 0\}$$

and

$$B = \{a^m b^m c^n : m \geq 0, n \geq 0\}.$$

- Prove that both A and B are context-free, by constructing two context-free grammars, one that generates A and one that generates B.
- We have seen in Section 3.8.2 that the language

  $$\{a^n b^n c^n : n \geq 0\}$$

  is not context-free. Explain why this implies that the intersection of two context-free languages is not necessarily context-free.
- Use De Morgan's Law to conclude that the complement of a context-free language is not necessarily context-free.

3.6 Let A be a context-free language and let B be a regular language.

- Prove that the intersection $A \cap B$ of A and B is context-free.
- Prove that the set-difference

  $$A \setminus B = \{w : w \in A, w \notin B\}$$

  of A and B is context-free.
- Is the set-difference of two context-free languages necessarily context-free?

3.7 Let L be the language consisting of all non-empty strings w over the alphabet $\{a, b\}$ such that

- the number of as in w is equal to the number of bs in w,
- w does not contain the substring $abba$, and
- w does not contain the substring $bbaa$.

In this exercise, you will prove that L is context-free.

Let A be the language consisting of all non-empty strings w over the alphabet $\{a,b\}$ such that the number of as in w is equal to the number of bs in w. In Exercise 3.3, you have shown that A is context-free.

Let B be the language consisting of all strings w over the alphabet $\{a,b\}$ such that

- w does not contain the substring $abba$, and
- w does not contain the substring $bbaa$.

1. Give a regular expression that describes the complement of B.
2. Argue that B is a regular language.
3. Use Exercise 3.6 to argue that L is a context-free language.

3.8 Construct (deterministic or nondeterministic) pushdown automata that accept the following languages.

1. $\{0^{2n}1^n : n \geq 0\}$.
2. $\{0^n1^m0^n : n \geq 1, m \geq 1\}$.
3. $\{w \in \{0,1\}^* :$ w contains more 1s than 0s$\}$.
4. $\{ww^R : w \in \{0,1\}^*\}$.
   (If $w = w_1 \ldots w_n$, then $w^R = w_n \ldots w_1$.)
5. $\{w \in \{0,1\}^* : w$ is a palindrome$\}$.

3.9 Let L be the language

$$L = \{a^m b^n : 0 \leq m \leq n \leq 2m\}.$$

1. Prove that L is context-free, by constructing a context-free grammar whose language is equal to L.
2. Prove that L is context-free, by constructing a nondeterministic pushdown automaton that accepts L.

3.10 Prove that the following languages are not context-free.

- $\{a^n\, b\, a^{2n}\, b\, a^{3n} : n \geq 0\}$.
- $\{a^n b^n a^n b^n : n \geq 0\}$.
- $\{a^m b^n c^k : m \geq 0, n \geq 0, k = \max(m, n)\}$.
- $\{w\#x : \ w$ is a substring of x, and $w, x \in \{a, b\}^*\}$.
  For example, the string $aba\#abbababbb$ is in the language, whereas the string $aba\#baabbaabb$ is not in the language. The alphabet is $\{a, b, \#\}$.
- $$\begin{array}{lll} \{\ w \in \{a, b, c\}^* & : & w \text{ contains more } b\text{'s than } a\text{'s and} \\ & & w \text{ contains more } c\text{'s than } a\text{'s }\}. \end{array}$$
- $\{1^n : \ n$ is a prime number$\}$.
- $\{(ab^n)^n : n \geq 0\}$. (The parentheses are not part of the alphabet; thus, the alphabet is $\{a, b, \}$.)

3.11 Let L be a language consisting of finitely many strings. Show that L is regular and, therefore, context-free. Let k be the maximum length of any string in L.

- Prove that *every* context-free grammar in Chomsky normal form that generates L has more than $\log k$ variables. (The logarithm is in base 2.)
- Prove that there is a context-free grammar that generates L and that has only one variable.

3.12 Let L be a context-free language. Prove that there exists an integer $p \geq 1$, such that the following is true: For every string s in L with $|s| \geq p$, there exists a string s' in L such that $|s| < |s'| \leq |s| + p$.

Chapter 4

Turing Machines and the Church-Turing Thesis

In the previous chapters, we have seen several computational devices that can be used to accept or generate regular and context-free languages. Even though these two classes of languages are fairly large, we have seen in Section 3.8.2 that these devices are not powerful enough to accept simple languages such as $A = \{a^m b^n c^{mn} : m \geq 0, n \geq 0\}$. In this chapter, we introduce the Turing machine, which is a simple model of a real computer. Turing machines can be used to accept all context-free languages, but also languages such as A. We will argue that every problem that can be solved on a real computer can also be solved by a Turing machine (this statement is known as the Church-Turing Thesis). In Chapter 5, we will consider the limitations of Turing machines and, hence, of real computers.

4.1 Definition of a Turing machine

We start with an informal description of a Turing machine. Such a machine consists of the following, see also Figure 4.1.

1. There are k *tapes*, for some fixed $k \geq 1$. Each tape is divided into *cells*, and is infinite both to the left and to the right. Each cell stores a symbol belonging to a finite set Γ, which is called the *tape alphabet*. The tape alphabet contains the *blank symbol* $\Box$. If a cell contains $\Box$, then this means that the cell is actually empty.

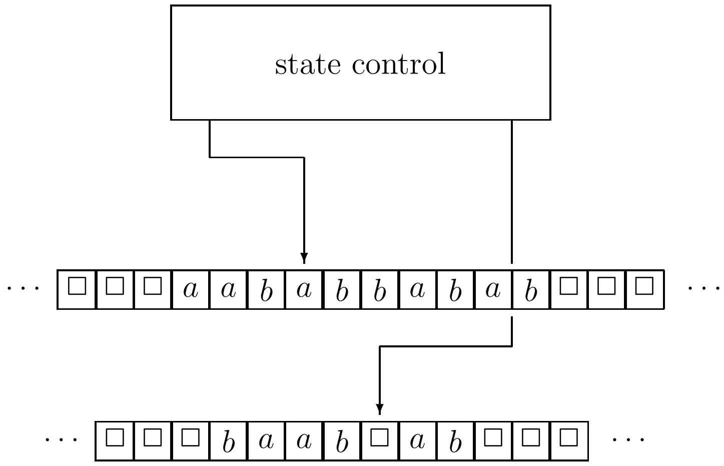


Figure 4.1: *A Turing machine with $k = 2$ tapes.*

2. Each tape has a *tape head* which can move along the tape, one cell per move. It can also read the cell it currently scans and replace the symbol in this cell by another symbol.
3. There is a *state control*, which can be in any one of a finite number of *states*. The finite set of states is denoted by Q. The set Q contains three special states: a *start state*, an *accept state*, and a *reject state*.

The Turing machine performs a sequence of *computation steps*. In one such step, it does the following:

1. Immediately before the computation step, the Turing machine is in a state r of Q, and each of the k tape heads is on a certain cell.
2. Depending on the current state r and the k symbols that are read by the tape heads,
   (a) the Turing machine switches to a state r' of Q (which may be equal to r),
   (b) each tape head writes a symbol of Γ in the cell it is currently scanning (this symbol may be equal to the symbol currently stored in the cell), and

(c) each tape head either moves one cell to the left, moves one cell to the right, or stays at the current cell.

We now give a formal definition of a deterministic Turing machine.

Definition 4.1.1 A *deterministic Turing machine* is a 7-tuple

$$M = (\Sigma, \Gamma, Q, \delta, q, q_{accept}, q_{reject}),$$

where

1. Σ is a finite set, called the *input alphabet*; the blank symbol $\Box$ is not contained in Σ,
2. Γ is a finite set, called the *tape alphabet*; this alphabet contains the blank symbol $\Box$, and $\Sigma \subseteq \Gamma$,
3. Q is a finite set, whose elements are called *states*,
4. q is an element of Q; it is called the *start state*,
5. q_{accept} is an element of Q; it is called the *accept state*,
6. q_{reject} is an element of Q; it is called the *reject state*,
7. δ is called the *transition function*, which is a function

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, N\}^k.$$

The transition function δ is basically the "program" of the Turing machine. This function tells us what the machine can do in "one computation step": Let $r \in Q$, and let $a_1, a_2, \ldots, a_k \in \Gamma$. Furthermore, let $r' \in Q$, $a'_1, a'_2, \ldots, a'_k \in \Gamma$, and $\sigma_1, \sigma_2, \ldots, \sigma_k \in \{L, R, N\}$ be such that

$$\delta(r, a_1, a_2, \ldots, a_k) = (r', a'_1, a'_2, \ldots, a'_k, \sigma_1, \sigma_2, \ldots, \sigma_k). \tag{4.1}$$

This transition means that if

- the Turing machine is in state r, and
- the head of the i-th tape reads the symbol a_i, $1 \leq i \leq k$,

then

- the Turing machine switches to state r',
- the head of the i-th tape replaces the scanned symbol a_i by the symbol a'_i, $1 \leq i \leq k$, and
- the head of the i-th tape moves according to σ_i, $1 \leq i \leq k$: if $\sigma_i = L$, then the tape head moves one cell to the left; if $\sigma_i = R$, then it moves one cell to the right; if $\sigma_i = N$, then the tape head does not move.

We will write the computation step (4.1) in the form of the *instruction*

$$ra_1a_2 \ldots a_k \rightarrow r'a'_1a'_2 \ldots a'_k\sigma_1\sigma_2 \ldots \sigma_k.$$

We now specify the computation of the Turing machine

$$M = (\Sigma, \Gamma, Q, \delta, q, q_{accept}, q_{reject}).$$

Start configuration: The input is a string over the input alphabet Σ. Initially, this input string is stored on the first tape, and the head of this tape is on the leftmost symbol of the input string. Initially, all other $k-1$ tapes are empty, i.e., only contain blank symbols, and the Turing machine is in the start state q.

Computation and termination: Starting in the start configuration, the Turing machine performs a sequence of computation steps as described above. The computation *terminates* at the moment when the Turing machine enters the accept state q_{accept} or the reject state q_{reject}. (Hence, if the Turing machine never enters the states q_{accept} and q_{reject}, the computation does *not* terminate.)

Acceptance: The Turing machine M *accepts* the input string $w \in \Sigma^*$, if the computation on this input terminates in the state q_{accept}. If the computation on this input terminates in the state q_{reject}, then M *rejects* the input string w.

We denote by $L(M)$ the language *accepted* by the Turing machine M. Thus, $L(M)$ is the set of all strings in Σ^* that are accepted by M.

Observe that a string $w \in \Sigma^*$ does not belong to $L(M)$ if and only if on input w,

- the computation of M terminates in the state q_{reject} or
- the computation of M does not terminate.

4.2 Examples of Turing machines

4.2.1 Accepting palindromes using one tape

We will show how to construct a Turing machine with one tape, that decides whether or not any input string $w \in \{a,b\}^*$ is a *palindrome*. Recall that the string w is called a palindrome, if reading w from left to right gives the same result as reading w from right to left. Examples of palindromes are *abba*, *baabbbbaab*, and the empty string ϵ.

Start of the computation: The tape contains the input string w, the tape head is on the leftmost symbol of w, and the Turing machine is in the start state q_0.

Idea: The tape head reads the leftmost symbol of w, deletes this symbol and "remembers" it by means of a state. Then the tape head moves to the rightmost symbol and tests whether it is equal to the (already deleted) leftmost symbol.

- If they are equal, then the rightmost symbol is deleted, the tape head moves to the new leftmost symbol, and the whole process is repeated.
- If they are not equal, the Turing machine enters the reject state, and the computation terminates.

The Turing machine enters the accept state as soon as the string currently stored on the tape is empty.

We will use the input alphabet $\Sigma = \{a,b\}$ and the tape alphabet $\Gamma = \{a,b,\Box\}$. The set Q of states consists of the following eight states:

q_0 :	start state; tape head is on the leftmost symbol
q_a :	leftmost symbol was a; tape head is moving to the right
q_b :	leftmost symbol was b; tape head is moving to the right
q'_a :	reached rightmost symbol; test whether it is equal to a, and delete it
q'_b :	reached rightmost symbol; test whether it is equal to b, and delete it
q_L :	test was positive; tape head is moving to the left
q_{accept} :	accept state
q_{reject} :	reject state

The transition function δ is specified by the following instructions:

$$
\begin{array}{lll}
q_0 a \to q_a \Box R & q_a a \to q_a a R & q_b a \to q_b a R \\
q_0 b \to q_b \Box R & q_a b \to q_a b R & q_b b \to q_b b R \\
q_0 \Box \to q_{accept} & q_a \Box \to q'_a \Box L & q_b \Box \to q'_b \Box L \\
 & & \\
q'_a a \to q_L \Box L & q'_b a \to q_{reject} & q_L a \to q_L a L \\
q'_a b \to q_{reject} & q'_b b \to q_L \Box L & q_L b \to q_L b L \\
q'_a \Box \to q_{accept} & q'_b \Box \to q_{accept} & q_L \Box \to q_0 \Box R
\end{array}
$$

You should go through the computation of this Turing machine for some sample inputs, for example $abba$, b, abb and the empty string (which is a palindrome).

4.2.2 Accepting palindromes using two tapes

We again consider the palindrome problem, but now we use a Turing machine with two tapes.

Start of the computation: The first tape contains the input string w and the head of the first tape is on the leftmost symbol of w. The second tape is empty and its tape head is at an arbitrary position. The Turing machine is in the start state q_0.

Idea: First, the input string w is copied to the second tape. Then the head of the first tape moves back to the leftmost symbol of w, while the head of the second tape stays at the rightmost symbol of w. Finally, the actual test starts: The head of the first tape moves to the right and, at the same time, the head of the second tape moves to the left. While moving, the Turing machine tests whether the two tape heads read the same symbol in each step.

The input alphabet is $\Sigma = \{a, b\}$ and the tape alphabet is $\Gamma = \{a, b, \Box\}$. The set Q of states consists of the following five states:

q_0 :	start state; copy w to the second tape
q_1 :	w has been copied; head of first tape moves to the left
q_2 :	head of first tape moves to the right; head of second tape moves to the left; until now, all tests were positive
q_{accept} :	accept state
q_{reject} :	reject state

The transition function δ is specified by the following instructions:

$$\begin{array}{ll}
q_0 a\Box \to q_0 aaRR & q_1 aa \to q_1 aaLN \\
q_0 b\Box \to q_0 bbRR & q_1 ab \to q_1 abLN \\
q_0 \Box\Box \to q_1 \Box\Box LL & q_1 ba \to q_1 baLN \\
 & q_1 bb \to q_1 bbLN \\
 & q_1 \Box a \to q_2 \Box aRN \\
 & q_1 \Box b \to q_2 \Box bRN \\
 & q_1 \Box\Box \to q_{accept}
\end{array}$$

$$\begin{array}{l}
q_2 aa \to q_2 aaRL \\
q_2 ab \to q_{reject} \\
q_2 ba \to q_{reject} \\
q_2 bb \to q_2 bbRL \\
q_2 \Box\Box \to q_{accept}
\end{array}$$

Again, you should run this Turing machine for some sample inputs.

4.2.3 Accepting $a^n b^n c^n$ using one tape

We will construct[1] a Turing machine with one tape that accepts the language

$$\{a^n b^n c^n : n \geq 0\}.$$

Recall that we have proved in Section 3.8.2 that this language is not context-free.

Start of the computation: The tape contains the input string w and the tape head is on the leftmost symbol of w. The Turing machine is in the start state.

Idea: In the previous examples, the tape alphabet Γ was equal to the union of the input alphabet Σ and $\{\Box\}$. In this example, we will add one symbol d to the tape alphabet. As we will see, this simplifies the construction of the Turing machine. Thus, the input alphabet is $\Sigma = \{a, b, c\}$ and the tape alphabet is $\Gamma = \{a, b, c, d, \Box\}$. Recall that the input string w belongs to Σ^*.

The general approach is to split the computation into two stages.

[1]Thanks to Michael Fleming for pointing out an error in a previous version of this construction.

Stage 1: In this stage, we check if the string w is in the language described by the regular expression $a^*b^*c^*$. If this is the case, then we walk back to the leftmost symbol. For this stage, we use the following states, besides the states q_{accept} and q_{reject}:

q_a : start state; we are reading the block of a's
q_b : we are reading the block of b's
q_c : we are reading the block of c's
q_L : walk to the leftmost symbol

Stage 2: In this stage, we repeat the following: Walk along the string from left to right, replace the leftmost a by d, replace the leftmost b by d, replace the leftmost c by d, and walk back to the leftmost symbol.

For this stage, we use the following states:

q'_a : start state of Stage 2; search for the leftmost a
q'_b : leftmost a has been replaced by d;
search for the leftmost b
q'_c : leftmost a has been replaced by d;
leftmost b has been replaced by d;
search for the leftmost c
q'_L : leftmost a has been replaced by d;
leftmost b has been replaced by d;
leftmost c has been replaced by d;
walk to the leftmost symbol

The transition function δ is specified by the following instructions:

$q_a a \rightarrow q_a a R$	$q_b a \rightarrow q_{reject}$
$q_a b \rightarrow q_b b R$	$q_b b \rightarrow q_b b R$
$q_a c \rightarrow q_c c R$	$q_b c \rightarrow q_c c R$
$q_a d \rightarrow$ cannot happen	$q_b d \rightarrow$ cannot happen
$q_a \Box \rightarrow q_L \Box L$	$q_b \Box \rightarrow q_L \Box L$
$q_c a \rightarrow q_{reject}$	$q_L a \rightarrow q_L a L$
$q_c b \rightarrow q_{reject}$	$q_L b \rightarrow q_L b L$
$q_c c \rightarrow q_c c R$	$q_L c \rightarrow q_L c L$
$q_c d \rightarrow$ cannot happen	$q_L d \rightarrow$ cannot happen
$q_c \Box \rightarrow q_L \Box L$	$q_L \Box \rightarrow q'_a \Box R$

$$
\begin{array}{ll}
q'_a a \rightarrow q'_b dR & q'_b a \rightarrow q'_b aR \\
q'_a b \rightarrow q_{reject} & q'_b b \rightarrow q'_c dR \\
q'_a c \rightarrow q_{reject} & q'_b c \rightarrow q_{reject} \\
q'_a d \rightarrow q'_a dR & q'_b d \rightarrow q'_b dR \\
q'_a \Box \rightarrow q_{accept} & q'_b \Box \rightarrow q_{reject}
\end{array}
$$

$$
\begin{array}{ll}
q'_c a \rightarrow q_{reject} & q'_L a \rightarrow q'_L aL \\
q'_c b \rightarrow q'_c bR & q'_L b \rightarrow q'_L bL \\
q'_c c \rightarrow q'_L dL & q'_L c \rightarrow q'_L cL \\
q'_c d \rightarrow q'_c dR & q'_L d \rightarrow q'_L dL \\
q'_c \Box \rightarrow q_{reject} & q'_L \Box \rightarrow q'_a \Box R
\end{array}
$$

We remark that Stage 1 is really necessary for this Turing machine: If we omit this stage, and use only Stage 2, then the string *aabcbc* will be accepted.

4.2.4 Accepting $a^n b^n c^n$ using tape alphabet $\{a, b, c, \Box\}$

We consider again the language $\{a^n b^n c^n : n \geq 0\}$. In the previous section, we presented a Turing machine that uses an extra symbol d. The reader may wonder if we can construct a Turing machine for this language that does not use any extra symbols. We will show below that this is indeed possible.

Start of the computation: The tape contains the input string w and the tape head is on the leftmost symbol of w. The Turing machine is in the start state q_0.

Idea: Repeat the following Stages 1 and 2, until the string is empty.

Stage 1. Walk along the string from left to right, delete the leftmost a, delete the leftmost b, and delete the rightmost c.

Stage 2. Shift the substring of bs and cs one position to the left; then walk back to the leftmost symbol.

The input alphabet is $\Sigma = \{a, b, c\}$ and the tape alphabet is $\Gamma = \{a, b, c, \Box\}$.

For Stage 1, we use the following states:

q_0 :	start state; tape head is on the leftmost symbol
q_a :	leftmost a has been deleted; have not read b
q_b :	leftmost b has been deleted; have not read c
q_c :	leftmost c has been read; tape head moves to the right
q_c' :	tape head is on the rightmost c
q_1 :	rightmost c has been deleted; tape head is on the rightmost symbol or $\Box$
q_{accept} :	accept state
q_{reject} :	reject state

The transitions for Stage 1 are specified by the following instructions:

$$\begin{array}{ll}
q_0 a \to q_a \Box R & q_a a \to q_a a R \\
q_0 b \to q_{reject} & q_a b \to q_b \Box R \\
q_0 c \to q_{reject} & q_a c \to q_{reject} \\
q_0 \Box \to q_{accept} & q_a \Box \to q_{reject} \\
 & \\
q_b a \to q_{reject} & q_c a \to q_{reject} \\
q_b b \to q_b b R & q_c b \to q_{reject} \\
q_b c \to q_c c R & q_c c \to q_c c R \\
q_b \Box \to q_{reject} & q_c \Box \to q_c' \Box L \\
 & q_c' c \to q_1 \Box L
\end{array}$$

For Stage 2, we use the following states:

q_1 :	as above; tape head is on the rightmost symbol or on $\Box$
q^c :	copy c one cell to the left
q^b :	copy b one cell to the left
q_2 :	done with shifting; head moves to the left

Additionally, we use a state q_1' which has the following meaning: If the input string is of the form $a^i bc$, for some $i \geq 1$, then after Stage 1, the tape contains the string $a^{i-1}\Box\Box$, the tape head is on the $\Box$ immediately to the right of the as, and the Turing machine is in state q_1. In this case, we move one cell to the left; if we then read $\Box$, then $i = 1$, and we accept; otherwise, we read a, and we reject.

The transitions for Stage 2 are specified by the following instructions:

$$
\begin{array}{ll}
q_1 a \to \text{cannot happen} & q_1' a \to q_{reject} \\
q_1 b \to q_{reject} & q_1' b \to \text{cannot happen} \\
q_1 c \to q^c \Box L & q_1' c \to \text{cannot happen} \\
q_1 \Box \to q_1' \Box L & q_1' \Box \to q_{accept}
\end{array}
$$

$$
\begin{array}{ll}
q^c a \to \text{cannot happen} & q^b a \to \text{cannot happen} \\
q^c b \to q^b c L & q^b b \to q^b b L \\
q^c c \to q^c c L & q^b c \to \text{cannot happen} \\
q^c \Box \to q_{reject} & q^b \Box \to q_2 b L
\end{array}
$$

$$
\begin{array}{l}
q_2 a \to q_2 a L \\
q_2 b \to \text{cannot happen} \\
q_2 c \to \text{cannot happen} \\
q_2 \Box \to q_0 \Box R
\end{array}
$$

4.2.5 Accepting $a^m b^n c^{mn}$ using one tape

We will sketch how to construct a Turing machine with one tape that accepts the language

$$\{a^m b^n c^{mn} : m \geq 0, n \geq 0\}.$$

Recall that we have proved in Section 3.8.2 that this language is not context-free.

The input alphabet is $\Sigma = \{a, b, c\}$ and the tape alphabet is $\Gamma = \{a, b, c, \$, \Box\}$, where the purpose of the symbol \$ will become clear below.

Start of the computation: The tape contains the input string w and the tape head is on the leftmost symbol of w. The Turing machine is in the start state.

Idea: Observe that a string $a^m b^n c^k$ is in the language if and only if for every a, the string contains n many cs. Based on this, the computation consists of the following stages:

Stage 1. Walk along the input string w from left to right and check whether w is an element of the language described by the regular expression $a^* b^* c^*$. If this is not the case, then reject the input string. Otherwise, go to Stage 2.

Stage 2. Walk back to the leftmost symbol of w. Go to Stage 3.

Stage 3. In this stage, the Turing machine does the following:

- Replace the leftmost a by the blank symbol $\Box$.
- Walk to the leftmost b.
- Zigzag between the bs and cs; each time, replace the leftmost b by the symbol \$, and replace the rightmost c by the blank symbol $\Box$. If, for some b, there is no c left, the Turing machine rejects the input string.
- Continue zigzagging until there are no bs left. Then go to Stage 4.

Observe that in this third stage, the string $a^m b^n c^k$ is transformed to the string $a^{m-1}\$^n c^{k-n}$.

Stage 4. In this stage, the Turing machine does the following:

- Replace each \$ by b.
- Walk to the leftmost a.

Hence, in this fourth stage, the string $a^{m-1}\$^n c^{k-n}$ is transformed to the string $a^{m-1} b^n c^{k-n}$.

Observe that the input string $a^m b^n c^k$ is in the language if and only if the string $a^{m-1} b^n c^{k-n}$ is in the language. Therefore, the Turing machine repeats Stages 3 and 4, until there are no as left. At that moment, it checks whether there are any cs left; if so, it rejects the input string; otherwise, it accepts the input string.

We hope that you believe that this description of the algorithm can be turned into a formal description of a Turing machine.

4.3 Multi-tape Turing machines

In Section 4.2, we have seen two Turing machines that accept palindromes; the first Turing machine has one tape, whereas the second one has two tapes. You will have noticed that the two-tape Turing machine was easier to obtain than the one-tape Turing machine. This leads to the question whether multi-tape Turing machines are more powerful than their one-tape counterparts. The answer is "no":

Theorem 4.3.1 *Let $k \geq 1$ be an integer. Any k-tape Turing machine can be converted to an equivalent one-tape Turing machine.*

Proof.[2] We will sketch the proof for the case when $k = 2$. Let $M = (\Sigma, \Gamma, Q, \delta, q, q_{accept}, q_{reject})$ be a two-tape Turing machine. Our goal is to convert M to an equivalent one-tape Turing machine N. That is, N should have the property that for all strings $w \in \Sigma^*$,

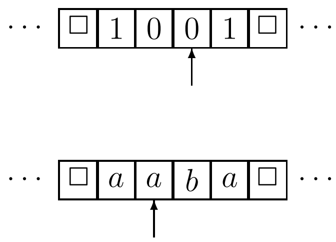
- M accepts w if and only if N accepts w,
- M rejects w if and only if N rejects w,
- M does not terminate on input w if and only if N does not terminate on input w.

The tape alphabet of the one-tape Turing machine N is

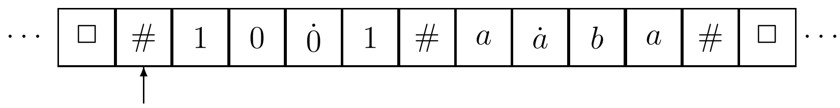
$$\Gamma \cup \{\dot{x} : x \in \Gamma\} \cup \{\#\}.$$

In words, we take the tape alphabet Γ of M, and add, for each $x \in \Gamma$, the symbol $\dot{x}$. Moreover, we add a special symbol #.

The Turing machine N will be defined in such a way that any configuration of the two-tape Turing machine M, for example



corresponds to the following configuration of the one-tape Turing machine N:



[2]Thanks to Sergio Cabello for pointing out an error in a previous version of this proof.

Thus, the contents of the two tapes of M are encoded on the single tape of N. The dotted symbols are used to indicate the positions of the two tape heads of M, whereas the three occurrences of the special symbol # are used to mark the boundaries of the strings on the two tapes of M.

The Turing machine N simulates one computation step of M, in the following way:

- Throughout the simulation of this step, N "remembers" the current state of M.

- At the start of the simulation, the tape head of N is on the leftmost symbol #.

- N walks along the string to the right until it finds the first dotted symbol. (This symbol indicates the location of the head on the first tape of M.) N remembers this first dotted symbol and continues walking to the right until it finds the second dotted symbol. (This symbol indicates the location of the head on the second tape of M.) Again, N remembers this second dotted symbol.

- At this moment, N is still at the second dotted symbol. N updates this part of the tape, by making the change that M would make on its second tape. (This change is given by the transition function of M; it depends on the current state of M and the two symbols that M reads on its two tapes.)

- N walks to the left until it finds the first dotted symbol. Then, it updates this part of the tape, by making the change that M would make on its first tape.

- In the previous two steps, in which the tape is updated, it may be necessary to shift a part of the tape.

- Finally, N remembers the new state of M and walks back to the leftmost symbol #.

It should be clear that the Turing machine N can be constructed by introducing appropriate states. ∎

4.4 The Church-Turing Thesis

We all have some intuitive notion of what an *algorithm* is. This notion will probably be something like "an algorithm is a procedure consisting of computation steps that can be specified in a finite amount of text". For example, any "computational process" that can be specified by a Java program, should be considered an algorithm. Similarly, a Turing machine specifies a "computational process" and, therefore, should be considered an algorithm. This leads to the question of whether it is possible to give a mathematical definition of an "algorithm". We just saw that every Java program represents an algorithm and that every Turing machine also represents an algorithm. Are these two notions of an algorithm equivalent? The answer is "yes". In fact, the following theorem states that many different notions of "computational process" are equivalent. (We hope that you have gained sufficient intuition, so that none of the claims in this theorem comes as a surprise to you.)

Theorem 4.4.1 *The following computation models are equivalent, i.e., any one of them can be converted to any other one:*

1. *One-tape Turing machines.*
2. *k-tape Turing machines, for any $k \geq 1$.*
3. *Non-deterministic Turing machines.*
4. *Java programs.*
5. *C++ programs.*
6. *Lisp programs.*

In other words, if we *define* the notion of an algorithm using any of the models in this theorem, then it does not matter which model we take: All these models give the same notion of an algorithm.

The problem of defining the notion of an algorithm goes back to David Hilbert. On August 8, 1900, at the Second International Congress of Mathematicians in Paris, Hilbert presented a list of problems that he considered crucial for the further development of mathematics. Hilbert's 10th problem is the following:

> Does there exist a *finite process* that decides whether or not any given polynomial with integer coefficients has integral roots?

Of course, in our language, Hilbert asked whether or not there exists an *algorithm* that decides, when given an arbitrary polynomial equation (with integer coefficients) such as

$$12x^3y^7z^5 + 7x^2y^4z - x^4 + y^2z^7 - z^3 + 10 = 0,$$

whether or not this equation has a solution in integers. In 1970, Matiyasevich proved that such an algorithm does *not* exist. Of course, in order to prove this claim, we first have to agree on what an *algorithm* is. In the beginning of the twentieth century, mathematicians gave several definitions, such as Turing machines (1936) and the λ-calculus (1936), and they proved that all these are equivalent. Later, after programming languages were invented, it was shown that these older notions of an algorithm are equivalent to notions of an algorithm that are based on C programs, Java programs, Lisp programs, Pascal programs, etc.

In other words, all attempts to give a rigorous definition of the notion of an algorithm led to the same concept. Because of this, computer scientists nowadays agree on what is called the Church-Turing Thesis:

Church-Turing Thesis: Every computational process that is intuitively considered to be an algorithm can be converted to a Turing machine.

In other words, this basically states that we *define* an algorithm to be a Turing machine. At this point, you should ask yourself, whether the Church-Turing Thesis can be *proved*. Alternatively, what has to be done in order to *disprove* this thesis?

Exercises

4.1 Construct a Turing machine with one tape, that accepts the language

$$\{0^{2n}1^n : n \geq 0\}.$$

Assume that, at the start of the computation, the tape head is on the leftmost symbol of the input string.

4.2 Construct a Turing machine with one tape, that accepts the language

$$\{w : \ w \text{ contains twice as many 0s as 1s}\}.$$

Assume that, at the start of the computation, the tape head is on the leftmost symbol of the input string.

4.3 Let A be the language

$$A = \{ w \in \{a,b,c\}^* : \begin{array}{l} w \text{ contains more } b\text{s than } a\text{s and} \\ w \text{ contains more } c\text{s than } a\text{s} \end{array} \}.$$

Give an informal description (in plain English) of a Turing machine with one tape, that accepts the language A.

4.4 Construct a Turing machine with one tape that receives as input a non-negative integer x and returns as output the integer $x + 1$. Integers are represented as binary strings.

Start of the computation: The tape contains the binary representation of the input x. The tape head is on the leftmost symbol and the Turing machine is in the start state q_0. For example, if $x = 431$, the tape looks as follows:

$\cdots$	$\Box$	$\Box$	$\Box$	1	1	0	1	0	1	1	1	1	$\Box$	$\Box$	$\Box$	$\cdots$
				$\uparrow$												

End of the computation: The tape contains the binary representation of the integer $x + 1$. The tape head is on the leftmost symbol and the Turing machine is in the final state q_1. For our example, the tape looks as follows:

$\cdots$	$\Box$	$\Box$	$\Box$	1	1	0	1	1	0	0	0	0	$\Box$	$\Box$	$\Box$	$\cdots$
				$\uparrow$												

The Turing machine in this exercise does not have an accept state or a reject state; instead, it has a final state q_1. As soon as state q_1 is entered, the Turing machine terminates. At termination, the contents of the tape is the output of the Turing machine.

4.5 Construct a Turing machine with two tapes that receives as input two non-negative integers x and y, and returns as output the integer $x + y$. Integers are represented as binary strings.

Start of the computation: The first tape contains the binary representation of x and its head is on the rightmost symbol of x. The second tape contains the binary representation of y and its head is on the rightmost bit of y. At the start, the Turing machine is in the start state q_0.

End of the computation: The first tape contains the binary representation of x and its head is on the rightmost symbol of x. The second tape contains the binary representation of the integer $x+y$ (thus, the integer y is "gone"). The head of the second tape is on the rightmost bit of $x + y$. The Turing machine is in the final state q_1.

4.6 Give an informal description (in plain English) of a Turing machine with one tape that receives as input two non-negative integers x and y, and returns as output the integer $x+y$. Integers are represented as binary strings. If you are an adventurous student, you may give a formal definition of your Turing machine.

4.7 Construct a Turing machine with one tape that receives as input an integer $x \geq 1$ and returns as output the integer $x-1$. Integers are represented in binary.

Start of the computation: The tape contains the binary representation of the input x. The tape head is on the rightmost symbol of x and the Turing machine is in the start state q_0.

End of the computation: The tape contains the binary representation of the integer $x - 1$. The tape head is on the rightmost bit of $x - 1$ and the Turing machine is in the final state q_1.

4.8 Give an informal description (in plain English) of a Turing machine with three tapes that receives as input two non-negative integers x and y, and returns as output the integer xy. Integers are represented as binary strings.

Start of the computation: The first tape contains the binary representation of x and its head is on the rightmost symbol of x. The second tape contains the binary representation of y and its head is on the rightmost symbol of y. The third tape is empty and its head is at an arbitrary location. The Turing machine is in the start state q_0.

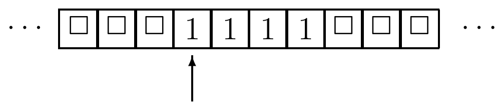
End of the computation: The first and second tapes are empty. The third tape contains the binary representation of the product xy and its head is on the rightmost bit of xy. The Turing machine is in the final state q_1.

Hint: Use the Turing machines of Exercises 4.5 and 4.7.

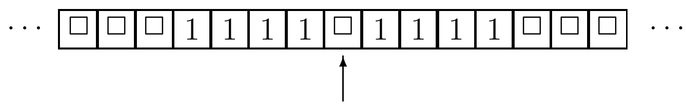
4.9 Construct a Turing machine with one tape that receives as input a string of the form 1^n for some integer $n \geq 0$; thus, the input is a string of n many 1s. The output of the Turing machine is the string $1^n \Box 1^n$. Thus, this Turing machine makes a copy of its input.

The input alphabet is $\Sigma = \{1\}$ and the tape alphabet is $\Gamma = \{1, \Box\}$.

Start of the computation: The tape contains a string of the form 1^n, for some integer $n \geq 0$, the tape head is on the leftmost symbol, and the Turing machine is in the start state. For example, if $n = 4$, the tape looks as follows:



End of the computation: The tape contains the string $1^n \Box 1^n$, the tape head is on the $\Box$ in the middle of this string, and the Turing machine is in the final state. For our example, the tape looks as follows:



The Turing machine in this exercise does not have an accept state or a reject state; instead, it has a final state. As soon as this state is entered, the Turing machine terminates. At termination, the contents of the tape is the output of the Turing machine.

Chapter 5

Decidable and Undecidable Languages

We have seen in Chapter 4 that Turing machines form a model for "everything that is intuitively computable". In this chapter, we consider the limitations of Turing machines. That is, we ask ourselves the question whether or not "everything" is computable. As we will see, the answer is "no". In fact, we will even see that "most" problems are not solvable by Turing machines and, therefore, not solvable by computers.

5.1 Decidability

In Chapter 4, we have defined when a Turing machine accepts an input string and when it rejects an input string. Based on this, we define the following class of languages.

Definition 5.1.1 Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. We say that A is *decidable*, if there exists a Turing machine M, such that for every string $w \in \Sigma^*$, the following holds:

1. If $w \in A$, then the computation of the Turing machine M, on the input string w, terminates in the accept state.

2. If $w \notin A$, then the computation of the Turing machine M, on the input string w, terminates in the reject state.

In other words, the language A is decidable, if there exists an algorithm that (i) terminates on every input string w, and (ii) correctly tells us whether $w \in A$ or $w \notin A$.

A language A that is not decidable is called *undecidable*. For such a language, there does *not* exist an algorithm that satisfies (i) and (ii) above.

In Section 4.2, we have seen several examples of languages that are decidable.

In the following subsections, we will give some examples of decidable and undecidable languages. These examples involve languages A whose elements are pairs of the form (C, w), where C is some computation model (for example, a deterministic finite automaton) and w is a string over the alphabet Σ. The pair (C, w) is in the language A if and only if the string w is in the language of the computation model C. For different computation models C, we will ask the question whether A is decidable, i.e., whether an algorithm exists that decides, for any input (C, w), whether or not this input belongs to the language A. Since the input to any algorithm is a string over some alphabet, we must encode the pair (C, w) as a string. In all cases that we consider, such a pair can be described using a finite amout of text. Therefore, we assume, without loss of generality, that binary strings are used for these encodings. Throughout the rest of this chapter, we will denote the binary encoding of a pair (C, w) by

$$\langle C, w \rangle.$$

5.1.1 The language A_{DFA}

We define the following language:

$$A_{DFA} = \{\langle M, w \rangle : \ M \text{ is a deterministic finite automaton that accepts the string } w\}.$$

Keep in mind that $\langle M, w \rangle$ denotes the binary string that forms an encoding of the finite automaton M and the string w that is given as input to M.

We claim that the language A_{DFA} is decidable. In order to prove this, we have to construct an algorithm with the following property, for any given input string u:

- If u is the encoding of a deterministic finite automaton M and a string w (i.e., u is in the correct format $\langle M, w \rangle$), and if M accepts w, then the algorithm terminates in its accept state.

- In all other cases, the algorithm terminates in its reject state.

An algorithm that exactly does this, is easy to obtain: On input u, the algorithm first checks whether or not u encodes a deterministic finite automaton M and a string w. If this is not the case, then it terminates and rejects the input string u. Otherwise, the algorithm "constructs" M and w, and then simulates the computation of M on the input string w. If M accepts w, then the algorithm terminates and accepts the input string u. If M does not accept w, then the algorithm terminates and rejects the input string u. Thus, we have proved the following result:

Theorem 5.1.2 *The language A_{DFA} is decidable.*

5.1.2 The language A_{NFA}

We define the following language:

$$A_{NFA} = \{\langle M, w\rangle : \ M \text{ is a nondeterministic finite automaton that accepts the string } w\}.$$

To prove that this language is decidable, consider the algorithm that does the following: On input u, the algorithm first checks whether or not u encodes a nondeterministic finite automaton M and a string w. If this is not the case, then it terminates and rejects the input string u. Otherwise, the algorithm constructs M and w. Since a computation of M (on input w) is not unique, the algorithm first converts M to an equivalent deterministic finite automaton N. Then, it proceeds as in Section 5.1.1.

Observe that the construction for converting a nondeterministic finite automaton to a deterministic finite automaton (see Section 2.5) is algorithmic, in the sense that it can be described by an algorithm. Because of this, the algorithm described above is a valid algorithm; it accepts all strings u that are in A_{NFA}, and it rejects all strings u that are not in A_{NFA}. Thus, we have proved the following result:

Theorem 5.1.3 *The language A_{NFA} is decidable.*

5.1.3 The language A_{CFG}

We define the following language:

$$A_{CFG} = \{\langle G, w\rangle : \ G \text{ is a context-free grammar such that } w \in L(G)\}.$$

We claim that this language is decidable. In order to prove this claim, consider a string u that encodes a context-free grammar $G = (V, \Sigma, S, R)$ and a string $w \in \Sigma^*$. Deciding whether or not $w \in L(G)$ is equivalent to deciding whether or not $S \stackrel{*}{\Rightarrow} w$. A first idea to decide this is by trying all possible derivations that start with the start variable S and that use rules of R. The problem is that, in case $w \notin L(G)$, it is not clear how many such derivations have to be checked before we can be sure that w is not in the language of G: If $w \in L(G)$, then it may be that w can be derived from S, only by first deriving a very long string, say v, and then use rules to shorten it so as to obtain the string w. Since there is no obvious upper bound on the length of the string v, we have to be careful.

The trick is to do the following. First, convert the grammar G to an equivalent grammar G' in Chomsky normal form. (The construction given in Section 3.4 can be described by an algorithm.) Let n be the length of the string w. Then, if $w \in L(G) = L(G')$, any derivation of w in G', from the start variable of G', consists of exactly $2n-1$ steps (where a "step" is defined as applying one rule of G'). Hence, we can decide whether or not $w \in L(G)$, by trying all possible derivations, in G', consisting of $2n-1$ steps. If one of these (finite number of) derivations leads to the string w, then $w \in L(G)$. Otherwise, $w \notin L(G)$. Thus, we have proved the following result:

Theorem 5.1.4 *The language A_{CFG} is decidable.*

In fact, the arguments above imply the following result:

Theorem 5.1.5 *Every context-free language is decidable.*

Proof. Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be an arbitrary context-free language. There exists a context-free grammar in Chomsky normal form, whose language is equal to A. Given an arbitrary string $w \in \Sigma^*$, we have seen above how we can decide whether or not w can be derived from the start variable of this grammar. ∎

5.1.4 The language A_{TM}

After having seen the languages A_{DFA}, A_{NFA}, and A_{CFG}, it is natural to consider the language

$$A_{TM} = \{\langle M, w\rangle : M \text{ is a Turing machine that accepts the string } w\}.$$

We will prove that this language is undecidable. Before we give the proof, let us mention what this means:

> There is no algorithm that, when given an arbitrary algorithm M and an arbitrary input string w for M, decides in a finite amount of time, whether or not M accepts w.

The proof of the claim that A_{TM} is undecidable is by contradiction. Thus, we assume that A_{TM} is decidable. Then there exists a Turing machine H that has the following property. For every input string $\langle M, w \rangle$ for H:

- If $\langle M, w \rangle \in A_{TM}$ (i.e., M accepts w), then H terminates in its accept state.
- If $\langle M, w \rangle \notin A_{TM}$ (i.e., M rejects w or M does not terminate on input w), then H terminates in its reject state.
- In particular, H terminates on any input $\langle M, w \rangle$.

We construct a new Turing machine D, that does the following: On input $\langle M \rangle$, the Turing machine D uses H as a subroutine to determine what M does when it is given its own description as input. Once D has determined this information, it does the *opposite* of what H does.

Turing machine D: On input $\langle M \rangle$, where M is a Turing machine, the new Turing machine D does the following:

Step 1: Run the Turing machine H on the input $\langle M, \langle M \rangle \rangle$.

Step 2:

- If H terminates in its accept state, then D terminates in its reject state.
- If H terminates in its reject state, then D terminates in its accept state.

First observe that this new Turing machine D terminates on any input string $\langle M \rangle$, because H terminates on every input. Next observe that, for any input string $\langle M \rangle$ for D:

- If $\langle M, \langle M \rangle \rangle \in A_{TM}$ (i.e., M accepts $\langle M \rangle$), then D terminates in its reject state.

- If $\langle M, \langle M \rangle \rangle \notin A_{TM}$ (i.e., M rejects $\langle M \rangle$ or M does not terminate on input $\langle M \rangle$), then D terminates in its accept state.

This means that for any string $\langle M \rangle$:

- If M accepts $\langle M \rangle$, then D rejects $\langle M \rangle$.
- If M rejects $\langle M \rangle$ or M does not terminate on input $\langle M \rangle$, then D accepts $\langle M \rangle$.

We now consider what happens if we give the Turing machine D the string $\langle D \rangle$ as input, i.e., we take $M = D$:

- If D accepts $\langle D \rangle$, then D rejects $\langle D \rangle$.
- If D rejects $\langle D \rangle$ or D does not terminate on input $\langle D \rangle$, then D accepts $\langle D \rangle$.

Since D terminates on every input string, this means that

- If D accepts $\langle D \rangle$, then D rejects $\langle D \rangle$.
- If D rejects $\langle D \rangle$, then D accepts $\langle D \rangle$.

This is clearly a contradiction. Therefore, the Turing machine H that decides the language A_{TM} cannot exist and, thus, A_{TM} is undecidable. We have proved the following result:

Theorem 5.1.6 *The language A_{TM} is undecidable.*

5.1.5 The Halting Problem

We define the following language:

$$Halt = \{\langle P, w \rangle : \quad P \text{ is a Java program that terminates on the input string } w\}.$$

Theorem 5.1.7 *The language Halt is undecidable.*

Proof. The proof is by contradiction. Thus, we assume that the language *Halt* is decidable. Then there exists a Java program H that takes as input a string of the form $\langle P, w \rangle$, where P is an arbitrary Java program and w is an arbitrary input for P. The program H has the following property:

- If $\langle P, w \rangle \in \textit{Halt}$ (i.e., program P terminates on input w), then H outputs *true*.
- If $\langle P, w \rangle \notin \textit{Halt}$ (i.e., program P does not terminate on input w), then H outputs *false*.
- In particular, H terminates on any input $\langle P, w \rangle$.

We will write the output of H as $H(P, w)$. Moreover, we will denote by $P(w)$ the computation obtained by running the program P on the input w. Hence,

$$H(P, w) = \begin{cases} \textit{true} & \text{if } P(w) \text{ terminates,} \\ \textit{false} & \text{if } P(w) \text{ does not terminate.} \end{cases}$$

Consider the following algorithm Q, which takes as input the encoding $\langle P \rangle$ of an arbitrary Java program P:

Algorithm $Q(\langle P \rangle)$**:**

while $H(P, \langle P \rangle) = \textit{true}$
do have a beer
endwhile

Since H is a Java program, this new algorithm Q can also be written as a Java program. Observe that

$Q(\langle P \rangle)$ terminates if and only if $H(P, \langle P \rangle) = \textit{false}$.

This means that for every Java program P,

$$Q(\langle P \rangle) \text{ terminates if and only if } P(\langle P \rangle) \text{ does not terminate.} \tag{5.1}$$

What happens if we run the Java program Q on the input string $\langle Q \rangle$? In other words, what happens if we run $Q(\langle Q \rangle)$? Then, in (5.1), we have to replace all occurrences of P by Q. Hence,

$Q(\langle Q \rangle)$ terminates if and only if $Q(\langle Q \rangle)$ does not terminate.

This is obviously a contradiction, and we can conclude that the Java program H does not exist. Therefore, the language *Halt* is undecidable. ∎

Remark 5.1.8 In this proof, we run the Java program Q on the input $\langle Q \rangle$. This means that the input to Q is a description of itself. In other words, we give Q itself as input. This is an example of what is called *self-reference*. Another example of self-reference can be found in Remark 5.1.8 of the textbook *Introduction to Theory of Computation* by A. Maheshwari and M. Smid.

5.2 Countable sets

The proofs that we gave in Sections 5.1.4 and 5.1.5 seem to be bizarre. In this section, we will convince you that these proofs in fact use a technique that you have seen in the course COMP 1805: *Cantor's Diagonalization*.

Let A and B be two sets and let $f : A \rightarrow B$ be a function. Recall that f is called a *bijection*, if

- f is *one-to-one* (or *injective*), i.e., for any two distinct elements a and a' in A, we have $f(a) \neq f(a')$, and
- f is *onto* (or *surjective*), i.e., for each element $b \in B$, there exists an element $a \in A$, such that $f(a) = b$.

The set of *natural numbers* is denoted by $\mathbb{N}$. That is, $\mathbb{N} = \{1, 2, 3, \ldots\}$.

Definition 5.2.1 Let A and B be two sets. We say that A and B have the *same size*, if there exists a bijection $f : A \rightarrow B$.

Definition 5.2.2 Let A be a set. We say that A is *countable*, if A is finite, or A and $\mathbb{N}$ have the same size.

In other words, if A is an infinite and countable set, then there exists a bijection $f : \mathbb{N} \rightarrow A$, and we can write A as

$$A = \{f(1), f(2), f(3), f(4), \ldots\}.$$

Since f is a bijection, every element of A occurs exactly once in the set on the right-hand side. This means that we can *number* the elements of A using the positive integers: Every element of A receives a unique number.

Theorem 5.2.3 *The following sets are countable:*

1. *The set* $\mathbb{Z}$ *of integers:*

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}.$$

2. *The Cartesian product* $\mathbb{N} \times \mathbb{N}$*:*

$$\mathbb{N} \times \mathbb{N} = \{(m, n) : m \in \mathbb{N}, n \in \mathbb{N}\}.$$

3. *The set* $\mathbb{Q}$ *of rational numbers:*

$$\mathbb{Q} = \{m/n : m \in \mathbb{Z}, n \in \mathbb{Z}, n \neq 0\}.$$

Proof. To prove that the set $\mathbb{Z}$ is countable, we have to give each element of $\mathbb{Z}$ a unique number in $\mathbb{N}$. We obtain this numbering, by listing the elements of $\mathbb{Z}$ in the following order:

$$0, 1, -1, 2, -2, 3, -3, 4, -4, \ldots$$

In this (infinite) list, every element of $\mathbb{Z}$ occurs exactly once. The number of an element of $\mathbb{Z}$ is given by its position in this list.

Formally, define the function $f : \mathbb{N} \to \mathbb{Z}$ by

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even,} \\ -(n-1)/2 & \text{if } n \text{ is odd.} \end{cases}$$

This function f is a bijection and, therefore, the sets $\mathbb{N}$ and $\mathbb{Z}$ have the same size. Hence, the set $\mathbb{Z}$ is countable.

For the proofs of the other two claims, we refer to the course COMP 1805. ∎

We now use Cantor's Diagonalization principle to prove that the set of real numbers is not countable:

Theorem 5.2.4 *The set* $\mathbb{R}$ *of real numbers is not countable.*

Proof. Define

$$A = \{x \in \mathbb{R} : 0 \leq x < 1\}.$$

We will prove that the set A is not countable. This will imply that the set $\mathbb{R}$ is not countable, because $A \subseteq \mathbb{R}$.

The proof that A is not countable is by contradiction. So we assume that A is countable. Then there exists a bijection $f : \mathbb{N} \rightarrow A$. Thus, for each $n \in \mathbb{N}$, $f(n)$ is a real number between zero and one. We can write

$$A = \{f(1), f(2), f(3), \ldots\}, \tag{5.2}$$

where every element of A occurs exactly once in the set on the right-hand side.

Consider the real number $f(1)$. We can write this number in decimal notation as

$$f(1) = 0.d_{11}d_{12}d_{13}\ldots,$$

where each d_{1i} is a digit in the set $\{0, 1, 2, \ldots, 9\}$. In general, for every $n \in \mathbb{N}$, we can write the real number $f(n)$ as

$$f(n) = 0.d_{n1}d_{n2}d_{n3}\ldots,$$

where, again, each d_{ni} is a digit in $\{0, 1, 2, \ldots, 9\}$.

We define the real number

$$x = 0.d_1d_2d_3\ldots,$$

where, for each integer $n \geq 1$,

$$d_n = \begin{cases} 4 & \text{if } d_{nn} \neq 4, \\ 5 & \text{if } d_{nn} = 4. \end{cases}$$

Observe that x is a real number between zero and one, i.e., $x \in A$. Therefore, by (5.2), there is an element $n \in \mathbb{N}$, such that $f(n) = x$. We compare the n-th digits of $f(n)$ and x:

- The n-th digit of $f(n)$ is equal to d_{nn}.
- The n-th digit of x is equal to d_n.

Since $f(n)$ and x are equal, their n-th digits must be equal, i.e., $d_{nn} = d_n$. But, by the definition of d_n, we have $d_{nn} \neq d_n$. This is a contradiction and, therefore, the set A is not countable.

Notice how we *defined* the real number x: For each $n \geq 1$, the n-th digit of x is *not* equal to the n-th digit of $f(n)$. Therefore, for each $n \geq 1$, $x \neq f(n)$ and, thus, $x \notin A$. ∎

The final result of this section is the fact that for every set A, its power set

$$\mathcal{P}(A) = \{B : B \subseteq A\}$$

is "strictly larger" than A. Define the function $f : A \to \mathcal{P}(A)$ by

$$f(a) = \{a\},$$

for any a in A. Since f is one-to-one, we can say that $\mathcal{P}(A)$ is "at least as large as" A.

Theorem 5.2.5 *Let A be an arbitrary set. Then A and $\mathcal{P}(A)$ do not have the same size.*

Proof. The proof is by contradiction. Thus, we assume that there exists a bijection $g : A \to \mathcal{P}(A)$. Define the set B as

$$B = \{a \in A : a \notin g(a)\}.$$

Since $B \in \mathcal{P}(A)$ and g is a bijection, there exists an element a in A such that $g(a) = B$.

First assume that $a \in B$. Since $g(a) = B$, we have $a \in g(a)$. But then, from the definition of the set B, we have $a \notin B$, which is a contradiction.

Next assume that $a \notin B$. Since $g(a) = B$, we have $a \notin g(a)$. But then, from the definition of the set B, we have $a \in B$, which is again a contradiction.

We conclude that the bijection g does not exist. Therefore, A and $\mathcal{P}(A)$ do not have the same size. ∎

5.2.1 The Halting Problem revisited

Now that we know about countability, we give a different way to look at the proof in Section 5.1.5 of the fact that the language

$$Halt = \{\langle P, w\rangle : \ P \text{ is a Java program that terminates on the input string } w\}$$

is undecidable. You should convince yourself that the proof given below follows the same reasoning as the one used in the proof of Theorem 5.2.4.

We first argue that the set of all Java programs is countable. Indeed, every Java program P can be described by a finite amount of text. In fact, we have been using $\langle P \rangle$ to denote such a description by a binary string. For any integer $n \geq 0$, there are at most 2^n (i.e., finitely many) Java programs P whose description $\langle P \rangle$ has length n. Therefore, to obtain a list of all Java programs, we do the following:

- List all Java programs P whose description $\langle P \rangle$ has length 0. (Well, the empty string does not describe any Java program, so in this step, nothing happens.)
- List all Java programs P whose description $\langle P \rangle$ has length 1.
- List all Java programs P whose description $\langle P \rangle$ has length 2.
- List all Java programs P whose description $\langle P \rangle$ has length 3.
- Etcetera, etcetera.

In this infinite list, every Java program occurs exactly once. Therefore, the set of all Java programs is countable.

Consider an infinite list

$$P_1, P_2, P_3, \ldots$$

in which every Java program occurs exactly once.

Assume that the language *Halt* is decidable. Then there exists a Java program H that decides this language. We may assume that, on input $\langle P, w \rangle$, H returns *true* if P terminates on input w, and *false* if P does not terminate on input w.

We construct a new Java program D that does the following:

Algorithm D: On input $\langle P_n \rangle$, where n is a positive integer, the new Java program D does the following:

Step 1: Run the Java program H on the input $\langle P_n, \langle P_n \rangle \rangle$.

Step 2:

- If H returns *true*, then D goes into an infinite loop.
- If H returns *false*, then D returns *true* and terminates its computation.

Observe that D can be written as a Java program. Therefore, there exists an integer $n \geq 1$ such that $D = P_n$. The next two observations follow from the pseudocode:

- If D terminates on input $\langle P_n \rangle$, then H returns *false* on input $\langle P_n, \langle P_n \rangle \rangle$, i.e., P_n does not terminate on input $\langle P_n \rangle$.

- If D does not terminate on input $\langle P_n \rangle$, then H returns *true* on input $\langle P_n, \langle P_n \rangle \rangle$, i.e., P_n terminates on input $\langle P_n \rangle$.

Thus,

- D terminates on input $\langle P_n \rangle$ if and only if P_n does not terminate on input $\langle P_n \rangle$.

Since $D = P_n$, this becomes

- D terminates on input $\langle D \rangle$ if and only if D does not terminate on input $\langle D \rangle$.

Thus, we have obtained a contradiction.

Remark 5.2.6 We *defined* the Java program D in such a way that, for each $n \geq 1$, the computation of D on input $\langle P_n \rangle$ differs from the computation of P_n on input $\langle P_n \rangle$. Hence, for each $n \geq 1$, $D \neq P_n$. However, since D is a Java program, there must be an integer $n \geq 1$ such that $D = P_n$.

5.3 Rice's Theorem

We have seen two examples of undecidable languages: A_{TM} and *Halt*. In this section, we prove that many languages involving Turing machines (or Java programs) are undecidable.

Define $\mathcal{T}$ to be the set of binary encodings of all Turing machines, i.e.,

$$\mathcal{T} = \{\langle M \rangle : M \text{ is a Turing machine with input alphabet } \{0,1\}\}.$$

Theorem 5.3.1 (Rice) *Let $\mathcal{P}$ be a subset of $\mathcal{T}$ such that*

1. *$\mathcal{P} \neq \emptyset$, i.e., there exists a Turing machine M such that $\langle M \rangle \in \mathcal{P}$,*

2. *$\mathcal{P}$ is a proper subset of $\mathcal{T}$, i.e., there exists a Turing machine N such that $\langle N \rangle \notin \mathcal{P}$, and*

3. for any two Turing machines M_1 and M_2 with $L(M_1) = L(M_2)$,

(a) either both $\langle M_1 \rangle$ and $\langle M_2 \rangle$ are in $\mathcal{P}$ or

(b) none of $\langle M_1 \rangle$ and $\langle M_2 \rangle$ is in $\mathcal{P}$.

Then the language $\mathcal{P}$ is undecidable.

You can think of $\mathcal{P}$ as the set of all Turing machines that satisfy a certain property. The first two conditions state that at least one Turing machine satisfies this property and not all Turing machines satisfy this property. The third condition states that, for any Turing machine M, whether or not M satisfies this property only depends on the language $L(M)$ of M.

Here are some examples of languages that satisfy the conditions in Rice's Theorem:

$$\mathcal{P}_1 = \{\langle M \rangle : M \text{ is a Turing machine and } \epsilon \in L(M)\},$$

$$\mathcal{P}_2 = \{\langle M \rangle : M \text{ is a Turing machine and } L(M) = \{1011, 001100\}\},$$

$$\mathcal{P}_3 = \{\langle M \rangle : M \text{ is a Turing machine and } L(M) \text{ is a regular language}\}.$$

You are encouraged to verify that Rice's Theorem indeed implies that each of $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ is undecidable.

5.3.1 Proof of Rice's Theorem

The strategy of the proof is as follows: Assuming that the language $\mathcal{P}$ is decidable, we show that the language

$$Halt = \{\langle M, w \rangle : \begin{array}{l} M \text{ is a Turing machine that terminates on} \\ \text{the input string } w\} \end{array}$$

is decidable. This will contradict Theorem 5.1.7.

The assumption that $\mathcal{P}$ is decidable implies the existence of a Turing machine H that decides $\mathcal{P}$. Observe that H takes as input a binary string $\langle M \rangle$ encoding a Turing machine M. In order to show that *Halt* is decidable, we need a Turing machine that takes as input a binary string $\langle M, w \rangle$ encoding a Turing machine M and a binary string w. In the rest of this section, we will explain how this Turing machine can be obtained.

Let M_1 be a Turing machine that, for any input string, switches in its first computation step from its start state to its reject state. In other words, M_1 is a Turing machine with $L(M_1) = \emptyset$. We assume that

$$\langle M_1 \rangle \notin \mathcal{P}.$$

(At the end of the proof, we will consider the case when $\langle M_1 \rangle \in \mathcal{P}$.) We also choose a Turing machine M_2 such that

$$\langle M_2 \rangle \in \mathcal{P}.$$

Consider a fixed Turing machine M and a fixed binary string w. We construct a new Turing machine T_{Mw} that takes as input an arbitrary binary string x:

Turing machine $T_{Mw}(x)$**:**

> run Turing machine M on input w;
> **if** M terminates
> **then** run M_2 on input x;
>   **if** M_2 terminates in the accept state
>   **then** terminate in the accept state
>   **else if** M_2 terminates in the reject state
>   **then** terminate in the reject state
>   **endif**
>   **endif**
> **endif**

We determine the language $L(T_{Mw})$ of this new Turing machine. In other words, we determine which strings x are accepted by T_{Mw}.

- Assume that M terminates on input w, i.e., $\langle M, w \rangle \in \mathit{Halt}$. Then it follows from the pseudocode that for any string x,

  $$x \text{ is accepted by } T_{Mw} \text{ if and only if } x \text{ is accepted by } M_2.$$

  Thus, $L(T_{Mw}) = L(M_2)$.

- Assume that M does not terminate on input w, i.e., $\langle M, w \rangle \notin \mathit{Halt}$. Then it follows from the pseudocode that for any string x, T_{Mw} does not terminate on input x. Thus, $L(T_{Mw}) = \emptyset$. In particular, $L(T_{Mw}) = L(M_1)$.

Recall that $\langle M_1 \rangle \notin \mathcal{P}$, whereas $\langle M_2 \rangle \in \mathcal{P}$. Then the following follows from the third condition in Rice's Theorem:

- If $\langle M, w \rangle \in \mathit{Halt}$, then $\langle T_{Mw} \rangle \in \mathcal{P}$.
- If $\langle M, w \rangle \notin \mathit{Halt}$, then $\langle T_{Mw} \rangle \notin \mathcal{P}$.

Thus, we have obtained a connection between the languages $\mathcal{P}$ and *Halt*. This suggests that we proceed as follows.

Assume that the language $\mathcal{P}$ is decidable. Let H be a Turing machine that decides $\mathcal{P}$. Then, for any Turing machine M,

- if $\langle M \rangle \in \mathcal{P}$, then H accepts the string $\langle M \rangle$,
- if $\langle M \rangle \notin \mathcal{P}$, then H rejects the string $\langle M \rangle$, and
- H terminates on any input string.

We construct a new Turing machine H' that takes as input an arbitrary string $\langle M, w \rangle$, where M is a Turing machine and w is a binary string:

Turing machine $H'(\langle M, w \rangle)$**:**

> construct the Turing machine T_{Mw} described above;
> run H on input $\langle T_{Mw} \rangle$;
> **if** H terminates in the accept state
> **then** terminate in the accept state
> **else** terminate in the reject state
> **endif**

It follows from the pseudocode that H' terminates on any input. We observe the following:

- Assume that $\langle M, w \rangle \in \mathit{Halt}$. Then we have seen before that $\langle T_{Mw} \rangle \in \mathcal{P}$. Since H decides the language $\mathcal{P}$, it follows that H accepts the string $\langle T_{Mw} \rangle$. Therefore, from the pseudocode, H' accepts the string $\langle M, w \rangle$.
- Assume that $\langle M, w \rangle \notin \mathit{Halt}$. Then we have seen before that $\langle T_{Mw} \rangle \notin \mathcal{P}$. Since H decides the language $\mathcal{P}$, it follows that H rejects (and terminates on) the string $\langle T_{Mw} \rangle$. Therefore, from the pseudocode, H' rejects (and terminates on) the string $\langle M, w \rangle$.

We have shown that the Turing machine H' decides the language *Halt*. This is a contradiction and, therefore, we conclude that the language $\mathcal{P}$ is undecidable.

Until now, we assumed that $\langle M_1 \rangle \notin \mathcal{P}$. If $\langle M_1 \rangle \in \mathcal{P}$, then we repeat the proof with $\mathcal{P}$ replaced by its complement $\overline{\mathcal{P}}$. This revised proof then shows that $\overline{\mathcal{P}}$ is undecidable. Since for every language L,

$$L \text{ is decidable if and only if } \overline{L} \text{ is decidable,}$$

we again conclude that $\mathcal{P}$ is undecidable.

5.4 Enumerability

We now come to the last class of languages in this chapter:

Definition 5.4.1 Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. We say that A is *enumerable*, if there exists a Turing machine M, such that for every string $w \in \Sigma^*$, the following holds:

1. If $w \in A$, then the computation of the Turing machine M, on the input string w, terminates in the accept state.
2. If $w \notin A$, then the computation of the Turing machine M, on the input string w, does not terminate in the accept state. That is, either the computation terminates in the reject state or the computation does not terminate.

In other words, the language A is enumerable, if there exists an algorithm having the following property. If $w \in A$, then the algorithm terminates on the input string w and tells us that $w \in A$. On the other hand, if $w \notin A$, then either (i) the algorithm terminates on the input string w and tells us that $w \notin A$ or (ii) the algorithm does not terminate on the input string w, in which case it does not tell us that $w \notin A$.

In Section 5.5, we will show where the term "enumerable" comes from. The following theorem follows immediately from Definitions 5.1.1 and 5.4.1.

Theorem 5.4.2 *Every decidable language is enumerable.*

In the following subsections, we will give some examples of enumerable languages.

5.4.1 Hilbert's problem

We have seen Hilbert's problem in Section 4.4: Is there an algorithm that decides, for any given polynomial p with integer coefficients, whether or not p has integral roots? If we formulate this problem in terms of languages, then Hilbert asked whether or not the language

$$\begin{aligned} Hilbert = \{\langle p \rangle : \quad & p \text{ is a polynomial with integer coefficients} \\ & \text{that has an integral root}\} \end{aligned}$$

is decidable. As usual, $\langle p \rangle$ denotes the binary string that forms an encoding of the polynomial p.

As we mentioned in Section 4.4, it was proven by Matiyasevich in 1970 that the language *Hilbert* is not decidable. We claim, that this language is enumerable. In order to prove this claim, we have to construct an algorithm HILBERT with the following property: For any input polynomial p with integer coefficients,

- if p has an integral root, then algorithm HILBERT will find one in a finite amount of time,
- if p does not have an integral root, then either algorithm HILBERT terminates and tells us that p does not have an integral root, or algorithm HILBERT does not terminate.

Recall that $\mathbb{Z}$ denotes the set of integers. Algorithm HILBERT does the following, on any input polynomial p with integer coefficients. Let n denote the number of variables in p. Algorithm HILBERT tries all elements $(x_1, x_2, \ldots, x_n) \in \mathbb{Z}^n$, in a systematic way, and for each such element, it computes $p(x_1, x_2, \ldots, x_n)$. If this value is zero, then algorithm HILBERT terminates and accepts the input.

We observe the following:

- If $p \in Hilbert$, then algorithm HILBERT terminates and accepts p, provided we are able to visit all elements $(x_1, x_2, \ldots, x_n) \in \mathbb{Z}^n$ in a "systematic way".
- If $p \notin Hilbert$, then $p(x_1, x_2, \ldots, x_n) \neq 0$ for all $(x_1, x_2, \ldots, x_n) \in \mathbb{Z}^n$ and, therefore, algorithm HILBERT does not terminate.

These are exactly the requirements for the language *Hilbert* to be enumerable.

It remains to explain how we visit all elements $(x_1, x_2, \ldots, x_n) \in \mathbb{Z}^n$ in a systematic way. For any integer $d \geq 0$, let H_d denote the hypercube in $\mathbb{Z}^n$ with sides of length $2d$ that is centered at the origin. That is, H_d consists of the set of all points $(x_1, x_2, \ldots, x_n)$ in $\mathbb{Z}^n$, such that $-d \leq x_i \leq d$ for all $1 \leq i \leq n$ and there exists at least one index j for which $x_j = d$ or $x_j = -d$. We observe that H_d contains a finite number of elements. In fact, if $d \geq 1$, then this number is equal to $(2d+1)^n - (2d-1)^n$. The algorithm will visit all elements $(x_1, x_2, \ldots, x_n) \in \mathbb{Z}^n$, in the following order: First, it visits the origin, which is the only element of H_0. Then, it visits all elements of H_1, followed by all elements of H_2, etc., etc.

To summarize, we obtain the following algorithm, proving that the language *Hilbert* is enumerable:

Algorithm HILBERT($\langle p \rangle$)**:**

```
n := the number of variables in p;
d := 0;
while d ≥ 0
do for each (x1, x2, ..., xn) ∈ Hd
    do R := p(x1, x2, ..., xn);
        if R = 0
        then terminate and accept
        endif
    endfor;
    d := d + 1
endwhile
```

Theorem 5.4.3 *The language Hilbert is enumerable.*

5.4.2 The language A_{TM}

We have shown in Section 5.1.4 that the language

$$A_{TM} = \{\langle M, w \rangle : M \text{ is a Turing machine that accepts the string } w\}.$$

is undecidable. In this section, we will prove that this language is enumerable. Thus, we have to construct an algorithm P having the following property, for any given input string u:

- If
  - u encodes a Turing machine M and an input string w for M (i.e., u is in the correct format $\langle M, w\rangle$) and
  - $\langle M, w\rangle \in A_{TM}$ (i.e., M accepts w),

  then algorithm P terminates in its accept state.
- In all other cases, either algorithm P terminates in its reject state, or algorithm P does not terminate.

On input string $u = \langle M, w\rangle$, which is in the correct format, algorithm P does the following:

1. It simulates the computation of M on input w.
2. If M terminates in its accept state, then P terminates in its accept state.
3. If M terminates in its reject state, then P terminates in its reject state.
4. If M does not terminate, then P does not terminate.

Hence, if $u = \langle M, w\rangle \in A_{TM}$, then M accepts w and, therefore, P accepts u. On the other hand, if $u = \langle M, w\rangle \notin A_{TM}$, then M does not accept w. This means that, on input w, M either terminates in its reject state or does not terminate. But this implies that, on input u, P either terminates in its reject state or does not terminate. This proves that algorithm P has the properties that are needed in order to show that the language A_{TM} is enumerable. We have proved the following result:

Theorem 5.4.4 *The language A_{TM} is enumerable.*

5.5 Where does the term "enumerable" come from?

In Definition 5.4.1, we have defined what it means for a language to be enumerable. In this section, we will see where this term comes from.

Definition 5.5.1 Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. An *enumerator* for A is a Turing machine E having the following properties:

1. Besides the standard features as in Section 4.1, E has a *print tape* and a *print state*. During its computation, E writes symbols of Σ on the print tape. Each time, E enters the print state, the current string on the print tape is *sent to the printer* and the print tape is made empty.

2. At the start of the computation, all tapes are empty and E is in the start state.

3. Every string w in A is sent to the printer at least once.

4. Every string w that is not in A is never sent to the printer.

Thus, an enumerator E for A really enumerates all strings in the language A. There is no particular order in which the strings of A are sent to the printer. Moreover, a string in A may be sent to the printer multiple times. If the language A is infinite, then the Turing machine E obviously does not terminate; however, every string in A (and only strings in A) will be sent to the printer at some time during the computation.

To give an example, let $A = \{0^n : n \geq 0\}$. The following Turing machine is an enumerator for A.

Turing machine StringsOfZeros**:**

> $n := 0$;
> **while** $1 + 1 = 2$
> **do for** $i := 1$ **to** n
>   **do** write 0 on the print tape
>   **endfor**;
>   enter the print state;
> $n := n + 1$
> **endwhile**

In the rest of this section, we will prove the following result.

Theorem 5.5.2 *A language is enumerable if and only if it has an enumerator.*

For the first part of the proof, assume that the language A has an enumerator E. We construct the following Turing machine M, which takes an arbitrary string w as input:

Turing machine $M(w)$**:**

> run E; every time E enters the print state:
> let v be the string on the print tape;
> **if** $w = v$
> **then** terminate in the accept state
> **endif**

The Turing machine M has the following properties:

- If $w \in A$, then w will be sent to the printer at some time during the computation of E. It follows from the pseudocode that, on input w, M terminates in the accept state.

- If $w \notin A$, then E will never sent w to the printer. It follows from the pseudocode that, on input w, M does not terminate.

Thus, M satisfies the conditions in Definition 5.4.1. We conclude that the language A is enumerable.

To prove the converse, we now assume that A is enumerable. Let M be a Turing machine that satisfies the conditions in Definition 5.4.1.

We fix an infinite list

$$s_1, s_2, s_3, \ldots$$

of all strings in Σ^*. For example, if $\Sigma = \{0, 1\}$, then we can take this list to be

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 100, 011, 101, 110, 111, \ldots$$

We construct the following Turing machine E, which takes the empty string as input:

Turing machine E:

```
n := 1;
while 1 + 1 = 2
do for i := 1 to n
    do run M for n steps on the input string s_i;
        if M accepts s_i within n steps
        then write s_i on the print tape;
                enter the print state
        endif
    endfor;
    n := n + 1
endwhile
```

We claim that E is an enumerator for the language A. To prove this, it is obvious that any string that is sent to the printer by E belongs to A.

It remains to prove that every string in A will be sent to the printer by E. Let w be a string in A. Then, on input w, the Turing machine M terminates in the accept state. Let m be the number of steps made by M on input w. Let i be the index such that $w = s_i$. Define $n = \max(m, i)$. Consider the n-th iteration of the while-loop and the i-th iteration of the for-loop. In this iteration, M accepts $s_i = w$ in $m \leq n$ steps and, therefore, w is sent to the printer.

5.6 Most languages are not enumerable

In this section, we will prove that most languages are not enumerable. The proof is based on the following two facts:

- The set consisting of all enumerable languages is countable; we will prove this in Section 5.6.1.

- The set consisting of all languages is not countable; we will prove this in Section 5.6.2.

5.6.1 The set of enumerable languages is countable

We define the set $\mathcal{E}$ as

$$\mathcal{E} = \{A : A \subseteq \{0,1\}^* \text{ is an enumerable language}\}.$$

In words, $\mathcal{E}$ is the set whose elements are the enumerable languages. Every element of $\mathcal{E}$ is an enumerable language. Hence, every element of the set $\mathcal{E}$ is itself a set consisting of strings.

Lemma 5.6.1 *The set $\mathcal{E}$ is countable.*

Proof. Let $A \subseteq \{0,1\}^*$ be an enumerable language. There exists a Turing machine T_A that satisfies the conditions in Definition 5.4.1. This Turing machine T_A can be uniquely specified by a string in English. This string can be converted to a binary string s_A. Hence, the binary string s_A is a unique encoding of the Turing machine T_A.

Consider the set

$$\mathcal{S} = \{s_A : A \subseteq \{0,1\}^* \text{ is an enumerable language}\}.$$

Observe that the function $f : \mathcal{E} \rightarrow \mathcal{S}$, defined by $f(A) = s_A$ for each $A \in \mathcal{E}$, is a bijection. Therefore, the sets $\mathcal{E}$ and $\mathcal{S}$ have the same size. Hence, in order to prove that the set $\mathcal{E}$ is countable, it is sufficient to prove that the set $\mathcal{S}$ is countable.

Why is the set $\mathcal{S}$ countable? For each integer $n \geq 0$, there are exactly 2^n binary strings of length n. Since there are binary strings that are not encodings of Turing machines, the set $\mathcal{S}$ contains at most 2^n strings of length n. In particular, the number of strings in $\mathcal{S}$ having length n is finite. Therefore, we obtain an infinite list of the elements of $\mathcal{S}$ in the following way:

- List all strings in $\mathcal{S}$ having length 0. (Well, the empty string is not in $\mathcal{S}$, so in this step, nothing happens.)
- List all strings in $\mathcal{S}$ having length 1.
- List all strings in $\mathcal{S}$ having length 2.
- List all strings in $\mathcal{S}$ having length 3.
- Etcetera, etcetera.

In this infinite list, every element of $\mathcal{S}$ occurs exactly once. Therefore, $\mathcal{S}$ is countable. ∎

5.6.2 The set of all languages is not countable

We define the set $\mathcal{L}$ as

$$\mathcal{L} = \{A : A \subseteq \{0,1\}^* \text{ is a language}\}.$$

In words, $\mathcal{L}$ is the set consisting of all languages. Every element of the set $\mathcal{L}$ is a set consisting of strings.

Lemma 5.6.2 *The set $\mathcal{L}$ is not countable.*

Proof. We define the set $\mathcal{B}$ as

$$\mathcal{B} = \{w : w \text{ is an infinite binary sequence}\}.$$

We claim that this set is not countable. The proof of this claim is almost identical to the proof of Theorem 5.2.4. We assume that the set $\mathcal{B}$ is countable. Then there exists a bijection $f : \mathbb{N} \to \mathcal{B}$. Thus, for each $n \in \mathbb{N}$, $f(n)$ is an infinite binary sequence. We can write

$$\mathcal{B} = \{f(1), f(2), f(3), \ldots\}, \tag{5.3}$$

where every element of $\mathcal{B}$ occurs exactly once in the set on the right-hand side.

We define the infinite binary sequence $w = w_1w_2w_3\ldots$, where, for each integer $n \geq 1$,

$$w_n = \begin{cases} 1 & \text{if the } n\text{-th bit of } f(n) \text{ is } 0, \\ 0 & \text{if the } n\text{-th bit of } f(n) \text{ is } 1. \end{cases}$$

Since $w \in \mathcal{B}$, it follows from (5.3) that there is an element $n \in \mathbb{N}$, such that $f(n) = w$. Hence, the n-th bits of $f(n)$ and w are equal. But, by definition, these n-th bits are not equal. This is a contradiction and, therefore, the set $\mathcal{B}$ is not countable.

In the rest of the proof, we will show that the sets $\mathcal{L}$ and $\mathcal{B}$ have the same size. Since $\mathcal{B}$ is not countable, this will imply that $\mathcal{L}$ is not countable.

In order to prove that $\mathcal{L}$ and $\mathcal{B}$ have the same size, we have to show that there exists a bijection

$$g : \mathcal{L} \to \mathcal{B}.$$

We first observe that the set $\{0,1\}^*$ is countable, because for each integer $n \geq 0$, there are only finitely many (to be precise, exactly 2^n) binary strings of length n. In fact, we can write

$$\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 100, 011, 101, 110, 111, \ldots\}.$$

For each integer $n \geq 1$, we denote by s_n the n-th string in this list. Hence,

$$\{0,1\}^* = \{s_1, s_2, s_3, \ldots\}. \tag{5.4}$$

Now we are ready to define the bijection $g : \mathcal{L} \rightarrow \mathcal{B}$: Let $A \in \mathcal{L}$, i.e., $A \subseteq \{0,1\}^*$ is a language. We define the infinite binary sequence $g(A)$ as follows: For each integer $n \geq 1$, the n-th bit of $g(A)$ is equal to

$$\begin{cases} 1 & \text{if } s_n \in A, \\ 0 & \text{if } s_n \notin A. \end{cases}$$

In words, the infinite binary sequence $g(A)$ contains a 1 exactly in those positions n for which the string s_n in (5.4) is in the language A.

To give an example, assume that A is the language consisting of all binary strings that start with 0. The following table gives the corresponding infinite binary sequence $g(A)$ (this sequence is obtained by reading the rightmost column from top to bottom):

$\{0,1\}^*$	A	$g(A)$
ϵ	not in A	0
0	in A	1
1	not in A	0
00	in A	1
01	in A	1
10	not in A	0
11	not in A	0
000	in A	1
001	in A	1
010	in A	1
100	not in A	0
011	in A	1
101	not in A	0
110	not in A	0
111	not in A	0
$\vdots$	$\vdots$	$\vdots$

The function g defined above has the following properties:

- If A and A' are two different languages in $\mathcal{L}$, then $g(A) \neq g(A')$.
- For every infinite binary sequence w in $\mathcal{B}$, there exists a language A in $\mathcal{L}$, such that $g(A) = w$.

This means that the function g is a bijection from $\mathcal{L}$ to $\mathcal{B}$. ∎

5.6.3 There are languages that are not enumerable

We have proved that the set

$$\mathcal{E} = \{A : A \subseteq \{0,1\}^* \text{ is an enumerable language}\}$$

is countable, whereas the set

$$\mathcal{L} = \{A : A \subseteq \{0,1\}^* \text{ is a language}\}$$

is not countable. This means that there are "more" languages in $\mathcal{L}$ than there are in $\mathcal{E}$, proving the following result:

Theorem 5.6.3 *There exist languages that are not enumerable.*

The proof given above shows the *existence* of languages that are not enumerable. However, the proof does not give us a specific example of a language that is not enumerable. In the next sections, we will see examples of such languages. Before we move on to these examples, we mention the difference between being countable and being enumerable:

- Any language A is countable, i.e., we can number the elements of A and, thus, write
$$A = \{s_1, s_2, s_3, s_4, \ldots\}.$$
- If the language A is enumerable, then, by Theorem 5.5.2, there is an algorithm that produces this numbering.
- If the language A is not enumerable, then, again by Theorem 5.5.2, there does not exist an algorithm that produces this numbering.

5.7 The relationship between decidable and enumerable languages

We know from Theorem 5.4.2 that every decidable language is enumerable. On the other hand, we know from Theorems 5.1.6 and 5.4.4 that the converse is not true. The following result should not come as a surprise:

Theorem 5.7.1 *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. Then, A is decidable if and only if both A and its complement $\overline{A}$ are enumerable.*

Proof. We first assume that A is decidable. Then, by Theorem 5.4.2, A is enumerable. Since A is decidable, it is not difficult to see that $\overline{A}$ is also decidable. Then, again by Theorem 5.4.2, $\overline{A}$ is enumerable.

To prove the converse, we assume that both A and $\overline{A}$ are enumerable. Since A is enumerable, there exists a Turing machine M_1, such that for any string $w \in \Sigma^*$, the following holds:

- If $w \in A$, then the computation of M_1, on the input string w, terminates in the accept state of M_1.

- If $w \notin A$, then the computation of M_1, on the input string w, terminates in the reject state of M_1 or does not terminate.

Similarly, since $\overline{A}$ is enumerable, there exists a Turing machine M_2, such that for any string $w \in \Sigma^*$, the following holds:

- If $w \in \overline{A}$, then the computation of M_2, on the input string w, terminates in the accept state of M_2.

- If $w \notin \overline{A}$, then the computation of M_2, on the input string w, terminates in the reject state of M_2 or does not terminate.

We construct a two-tape Turing machine M:

Two-tape Turing machine M: For any input string $w \in \Sigma^*$, M does the following:

- M simulates the computation of M_1, on input w, on the first tape, and, simultaneously, it simulates the computation of M_2, on input w, on the second tape.
- If the simulation of M_1 terminates in the accept state of M_1, then M terminates in its accept state.
- If the simulation of M_2 terminates in the accept state of M_2, then M terminates in its reject state.

Observe the following:

- If $w \in A$, then M_1 terminates in its accept state and, therefore, M terminates in its accept state.
- If $w \notin A$, then M_2 terminates in its accept state and, therefore, M terminates in its reject state.

We conclude that the Turing machine M accepts all strings in A, and rejects all strings that are not in A. This proves that the language A is decidable. ∎

We now use Theorem 5.7.1 to give examples of languages that are not enumerable:

Theorem 5.7.2 *The language $\overline{A_{TM}}$ is not enumerable.*

Proof. We know from Theorems 5.4.4 and 5.1.6 that the language A_{TM} is enumerable but not decidable. Combining these facts with Theorem 5.7.1 implies that the language $\overline{A_{TM}}$ is not enumerable. ∎

The following result can be proved in exactly the same way:

Theorem 5.7.3 *The language $\overline{Halt}$ is not enumerable.*

5.8 A language A such that both A and $\overline{A}$ are not enumerable

In Theorem 5.7.2, we have seen that the complement of the language A_{TM} is not enumerable. In Theorem 5.4.4, however, we have shown that the language A_{TM} itself is enumerable. In this section, we consider the language

$$EQ_{TM} = \{\langle M_1, M_2\rangle : \begin{array}{l} M_1 \text{ and } M_2 \text{ are Turing machines} \\ \text{and } L(M_1) = L(M_2)\}. \end{array}$$

We will show the following result:

Theorem 5.8.1 *Both EQ_{TM} and its complement $\overline{EQ}_{TM}$ are not enumerable.*

5.8.1 EQ_{TM} is not enumerable

Consider a fixed Turing machine M and a fixed binary string w. We construct a new Turing machine T_{Mw} that takes as input an arbitrary binary string x:

Turing machine $T_{Mw}(x)$:

run Turing machine M on input w;
terminate in the accept state

We determine the language $L(T_{Mw})$ of this new Turing machine. In other words, we determine which strings x are accepted by T_{Mw}.

- Assume that M terminates on input w, i.e., $\langle M, w\rangle \in \textit{Halt}$. Then it follows from the pseudocode that every string x is accepted by T_{Mw}. Thus, $L(T_{Mw}) = \{0,1\}^*$.

- Assume that M does not terminate on input w, i.e., $\langle M, w\rangle \in \overline{\textit{Halt}}$. Then it follows from the pseudocode that, for any string x, T_{Mw} does not terminate on input x. Thus, $L(T_{Mw}) = \emptyset$.

Assume that the language EQ_{TM} is enumerable. We will show that $\overline{\textit{Halt}}$ is enumerable as well, which will contradict Theorem 5.7.3.

Since EQ_{TM} is enumerable, there exists a Turing machine H having the following property, for any two Turing machines M_1 and M_2:

- If $L(M_1) = L(M_2)$, then, on input $\langle M_1, M_2 \rangle$, H terminates in the accept state.

- If $L(M_1) \neq L(M_2)$, then, on input $\langle M_1, M_2 \rangle$, H either terminates in the reject state or does not terminate.

We construct a new Turing machine H' that takes as input an arbitrary string $\langle M, w \rangle$, where M is a Turing machine and w is a binary string:

Turing machine $H'(\langle M, w \rangle)$**:**

> construct a Turing machine M_1 that rejects every input string;
> construct the Turing machine T_{Mw} described above;
> run H on input $\langle M_1, T_{Mw} \rangle$;
> **if** H terminates in the accept state
> **then** terminate in the accept state
> **else if** H terminates in the reject state
>   **then** terminate in the reject state
>   **endif**
> **endif**

We observe the following:

- Assume that $\langle M, w \rangle \in \overline{Halt}$. Then we have seen before that $L(T_{Mw}) = \emptyset$. By our choice of M_1, we have $L(M_1) = \emptyset$ as well. Therefore, H accepts (and terminates on) the input $\langle M_1, T_{Mw} \rangle$. It follows from the pseudocode that H' accepts (and terminates on) the string $\langle M, w \rangle$.

- Assume that $\langle M, w \rangle \notin \overline{Halt}$, i.e., $\langle M, w \rangle \in Halt$. Then we have seen before that $L(T_{Mw}) \neq \emptyset = L(M_1)$. Therefore, on input $\langle M_1, T_{Mw} \rangle$, H either terminates in the reject state or does not terminate. It follows from the pseudocode that, on input $\langle M, w \rangle$, H' either terminates in the reject state or does not terminate.

Thus, the Turing machine H' has the properties needed to show that the language $\overline{Halt}$ is enumerable. This is a contradiction and, therefore, we conclude that the language EQ_{TM} is not enumerable.

5.8.2 $\overline{EQ}_{TM}$ is not enumerable

This proof is symmetric to the one in Section 5.8.1. For a fixed Turing machine M and a fixed binary string w, we will use the same Turing machine T_{Mw} as in Section 5.8.1.

Assume that the language $\overline{EQ}_{TM}$ is enumerable. We will show that $\overline{Halt}$ is enumerable as well, which will contradict Theorem 5.7.3.

Since $\overline{EQ}_{TM}$ is enumerable, there exists a Turing machine H having the following property, for any two Turing machines M_1 and M_2:

- If $L(M_1) \neq L(M_2)$, then, on input $\langle M_1, M_2 \rangle$, H terminates in the accept state.
- If $L(M_1) = L(M_2)$, then, on input $\langle M_1, M_2 \rangle$, H either terminates in the reject state or does not terminate.

We construct a new Turing machine H' that takes as input an arbitrary string $\langle M, w \rangle$, where M is a Turing machine and w is a binary string:

Turing machine $H'(\langle M, w \rangle)$**:**

construct a Turing machine M_1 that accepts every input string;
construct the Turing machine T_{Mw} of Section 5.8.1;
run H on input $\langle M_1, T_{Mw} \rangle$;
if H terminates in the accept state
then terminate in the accept state
else if H terminates in the reject state
  then terminate in the reject state
  endif
endif

We observe the following:

- Assume that $\langle M, w \rangle \in \overline{Halt}$. Then we have seen before that $L(T_{Mw}) = \emptyset$. Thus, by our choice of M_1, we have $L(T_{Mw}) \neq L(M_1)$. Therefore, H accepts (and terminates on) the input $\langle M_1, T_{Mw} \rangle$. It follows from the pseudocode that H' accepts (and terminates on) the string $\langle M, w \rangle$.
- Assume that $\langle M, w \rangle \notin \overline{Halt}$. Then $L(T_{Mw}) = \{0, 1\}^* = L(M_1)$ and, on input $\langle M_1, T_{Mw} \rangle$, H either terminates in the reject state or does not

terminate. It follows from the pseudocode that, on input $\langle M, w \rangle$, H' either terminates in the reject state or does not terminate.

Thus, the Turing machine H' has the properties needed to show that the language $\overline{Halt}$ is enumerable. This is a contradiction and, therefore, we conclude that the language $\overline{EQ}_{TM}$ is not enumerable.

Exercises

5.1 Prove that the language

$$\{w \in \{0,1\}^* : w \text{ is the binary representation of } 2^n \text{ for some } n \geq 0\}$$

is decidable. In other words, construct a Turing machine that gets as input an arbitrary number $x \in \mathbb{N}$, represented in binary as a string w, and that decides whether or not x is a power of two.

5.2 Let F be the set of all functions $f : \mathbb{N} \to \mathbb{N}$. Prove that F is not countable.

5.3 A function $f : \mathbb{N} \to \mathbb{N}$ is called *computable*, if there exists a Turing machine, that gets as input an arbitrary positive integer n, written in binary, and gives as output the value of $f(n)$, again written in binary. This Turing machine has a final state. As soon as the Turing machine enters this final state, the computation terminates, and the output is the binary string that is written on its tape.

Prove that there exist functions $f : \mathbb{N} \to \mathbb{N}$ that are not computable.

5.4 Let n be a fixed positive integer, and let k be the number of bits in the binary representation of n. (Hence, $k = 1 + \lfloor \log n \rfloor$.) Construct a Turing machine with one tape, tape alphabet $\{0, 1, \Box\}$, and exactly $k + 1$ states $q_0, q_1, \ldots, q_k$, that does the following:

Start of the computation: The tape is empty, i.e., every cell of the tape contains $\Box$, and the Turing machine is in the start state q_0.

End of the computation: The tape contains the binary representation of the integer n, the tape head is on the rightmost bit of the binary representation of n, and the Turing machine is in the final state q_k.

The Turing machine in this exercise does not have an accept state or a reject state; instead, it has a final state q_k. As soon as state q_k is entered, the Turing machine terminates.

5.5 Give an informal description (in plain English) of a Turing machine with three tapes, that gets as input the binary representation of an arbitrary integer $m \geq 1$, and returns as output the unary representation of m.

Start of the computation: The first tape contains the binary representation of the input m. The other two tapes are empty (i.e., contain only $\Box$s). The Turing machine is in the start state.

End of the computation: The third tape contains the unary representation of m, i.e., a string consisting of m many ones. The Turing machine is in the final state.

The Turing machine in this exercise does not have an accept state or a reject state; instead, it has a final state. As soon as this final state is entered, the Turing machine terminates.

Hint: Use the second tape to maintain a string of ones, whose length is a power of two.

5.6 In this exercise, you are asked to prove that the *busy beaver* function $BB : \mathbb{N} \to \mathbb{N}$ is not computable.

For any integer $n \geq 1$, we define TM_n to be the set of all Turing machines M, such that

- M has one tape,
- M has exactly n states,
- the tape alphabet of M is $\{0, 1, \Box\}$, and
- M terminates, when given the empty string ϵ as input.

For every Turing machine $M \in TM_n$, we define $f(M)$ to be the number of ones on the tape, after the computation of M, on the empty input string, has terminated.

The busy beaver function $BB : \mathbb{N} \to \mathbb{N}$ is defined as

$$BB(n) := \max\{f(M) : M \in TM_n\}, \text{ for every } n \geq 1.$$

In words, $BB(n)$ is the maximum number of ones that any Turing machine with n states can produce, when given the empty string as input, and assuming the Turing machine terminates on this input.

Prove that the function BB is not computable.

Hint: Assume that BB is computable. Then there exists a Turing machine M that, for any given $n \geq 1$, computes the value of $BB(n)$. Fix a large integer $n \geq 1$. Define (in plain English) a Turing machine that, when given the empty string as input, terminates and outputs a string consisting of more than $BB(n)$ many ones. Use Exercises 5.4 and 5.5 to argue that there exists such a Turing machine having $O(\log n)$ states. Then, if you assume that n is large enough, the number of states is at most n.

5.7 Since the set

$$\mathcal{T} = \{M : M \text{ is a Turing machine}\}$$

is countable, there is an infinite list

$$M_1, M_2, M_3, M_4, \ldots,$$

such that every Turing machine occurs exactly once in this list.

For any positive integer n, let $\langle n \rangle$ denote the binary representation of n; observe that $\langle n \rangle$ is a binary string.

Let A be the language defined as

$$A = \{\langle n \rangle : \begin{array}{l}\text{the Turing machine } M_n \text{ terminates on the input string } \langle n \rangle, \\ \text{and it rejects this string}\}.\end{array}$$

Prove that the language A is undecidable.

5.8 Consider the three languages

$$\mathit{Empty} = \{\langle M \rangle : M \text{ is a Turing machine for which } L(M) = \emptyset\},$$

$$\mathit{UselessState} = \{\langle M, q \rangle : \begin{array}{l} M \text{ is a Turing machine, } q \text{ is a state of } M, \\ \text{for every input string } w, \text{ the computation of } M \text{ on} \\ \text{input } w \text{ never visits state } q\}, \end{array}$$

and

$$EQ_{TM} = \{\langle M_1, M_2 \rangle : \begin{array}{l} M_1 \text{ and } M_2 \text{ are Turing machines} \\ \text{and } L(M_1) = L(M_2)\}. \end{array}$$

- Use Rice's Theorem to show that *Empty* is undecidable.
- Use the first part to show that *UselessState* is undecidable.
- Use the first part to show that EQ_{TM} is undecidable.

5.9 Consider the language

$REG_{TM} = \{\langle M \rangle : M$ is a Turing machine whose language $L(M)$ is regular$\}$.

Use Rice's Theorem to prove that REG_{TM} is undecidable.

5.10 We have seen in Section 5.1.4 that the language

$$A_{TM} = \{\langle M, w \rangle : M \text{ is a Turing machine that accepts } w\}$$

is undecidable. Consider the language REG_{TM} of Exercise 5.9. The questions below will lead you through a proof of the claim that the language REG_{TM} is undecidable.

Consider a fixed Turing machine M and a fixed binary string w. We construct a new Turing machine T_{Mw} that takes as input an arbitrary binary string x:

Turing machine $T_{Mw}(x)$:

```
if x = 0^n 1^n for some n ≥ 0
then terminate in the accept state
else run M on the input string w;
     if M terminates in the accept state
     then terminate in the accept state
     else if M terminates in the reject state
          then terminate in the reject state
          endif
     endif
endif
```

Answer the following two questions:

- Assume that M accepts the string w. What is the language $L(T_{Mw})$ of the new Turing machine T_{Mw}?

- Assume that M does not accept the string w. What is the language $L(T_{Mw})$ of the new Turing machine T_{Mw}?

The goal is to prove that the language REG_{TM} is undecidable. We will prove this by contradiction. Thus, we assume that R is a Turing machine that decides REG_{TM}. Recall what this means:

- If M is a Turing machine whose language is regular, then R, when given $\langle M \rangle$ as input, will terminate in the accept state.
- If M is a Turing machine whose language is not regular, then R, when given $\langle M \rangle$ as input, will terminate in the reject state.

We construct a new Turing machine R' which takes as input an arbitrary Turing machine M and an arbitrary binary string w:

Turing machine $R'(\langle M, w \rangle)$**:**

> construct the Turing machine T_{Mw} described above;
> run R on the input $\langle T_{Mw} \rangle$;
> **if** R terminates in the accept state
> **then** terminate in the accept state
> **else if** R terminates in the reject state
>   **then** terminate in the reject state
>   **endif**
> **endif**

Prove that M accepts w if and only if R' (when given $\langle M, w \rangle$ as input), terminates in the accept state.

Now finish the proof by arguing that the language REG_{TM} is undecidable.

5.11 A Java program P is called a *Hello-World-program*, if the following is true: When given the empty string ϵ as input, P outputs the string `Hello World` and then terminates. (We do not care what P does when the input string is non-empty.)

Consider the language

$$HW = \{\langle P \rangle : \ P \text{ is a Hello-World-program}\}.$$

The questions below will lead you through a proof of the claim that the language HW is undecidable.

Consider a fixed Java program P and a fixed binary string w. We write a new Java program J_{Pw} which takes as input an arbitrary binary string x:

Java program $J_{Pw}(x)$**:**

> run P on the input w;
> print `Hello World`

- Argue that P terminates on input w if and only if $\langle J_{Pw} \rangle \in HW$.

The goal is to prove that the language HW is undecidable. We will prove this by contradiction. Thus, we assume that H is a Java program that decides HW. Recall what this means:

- If P is a Hello-World-program, then H, when given $\langle P \rangle$ as input, will terminate in the accept state.
- If P is not a Hello-World-program, then H, when given $\langle P \rangle$ as input, will terminate in the reject state.

We write a new Java program H' which takes as input the binary encoding $\langle P, w \rangle$ of an arbitrary Java program P and an arbitrary binary string w:

Java program $H'(\langle P, w \rangle)$**:**

> construct the Java program J_{Pw} described above;
> run H on the input $\langle J_{Pw} \rangle$;
> **if** H terminates in the accept state
> **then** terminate in the accept state
> **else** terminate in the reject state
> **endif**

Argue that the following are true:

- For any input $\langle P, w \rangle$, H' terminates.
- If P terminates on input w, then H' (when given $\langle P, w \rangle$ as input), terminates in the accept state.
- If P does not terminate on input w, then H' (when given $\langle P, w \rangle$ as input), terminates in the reject state.

Now finish the proof by arguing that the language HW is undecidable.

5.12 Prove that the language *Halt*, see Section 5.1.5, is enumerable.

5.13 We define the following language:

$$\begin{aligned} L = \{u \quad : \quad & u = \langle 0, M, w\rangle \text{ for some } \langle M, w\rangle \in A_{TM}, \\ & \text{or } u = \langle 1, M, w\rangle \text{ for some } \langle M, w\rangle \notin A_{TM} \ \} . \end{aligned}$$

Prove that neither L nor its complement $\overline{L}$ is enumerable.

Hint: There are two ways to solve this exercise. In the first solution, (i) you assume that L is enumerable, and then prove that A_{TM} is decidable, and (ii) you assume that $\overline{L}$ is enumerable, and then prove that A_{TM} is decidable. In the second solution, (i) you assume that L is enumerable, and then prove that $\overline{A_{TM}}$ is enumerable, and (ii) you assume that $\overline{L}$ is enumerable, and then prove that $\overline{A_{TM}}$ is enumerable.

Chapter 6

Complexity Theory

In the previous chapters, we have considered the problem of what can be computed by Turing machines (i.e., computers) and what cannot be computed. We did not, however, take the efficiency of the computations into account. In this chapter, we introduce a classification of decidable languages A, based on the running time of the "best" algorithm that decides A. That is, given a decidable language A, we are interested in the "fastest" algorithm that, for any given string w, decides whether or not $w \in A$.

6.1 The running time of algorithms

Let M be a Turing machine, and let w be an input string for M. We define the *running time* $t_M(w)$ of M on input w as

$$t_M(w) := \text{ the number of computation steps made by } M \text{ on input } w.$$

As usual, we denote by $|w|$, the number of symbols in the string w. We denote the set of non-negative integers by $\mathbb{N}_0$.

Definition 6.1.1 Let Σ be an alphabet, let $T : \mathbb{N}_0 \to \mathbb{N}_0$ be a function, let $A \subseteq \Sigma^*$ be a decidable language, and let $F : \Sigma^* \to \Sigma^*$ be a computable function.

- We say that the Turing machine M decides the language A *in time* T, if
$$t_M(w) \leq T(|w|)$$
for all strings w in Σ^*.

- We say that the Turing machine M computes the function F *in time* T, if
$$t_M(w) \leq T(|w|)$$
for all strings $w \in \Sigma^*$.

In other words, the "running time function" T is a function of the *length of the input*, which we usually denote by n. For any n, the value of $T(n)$ is an upper bound on the running time of the Turing machine M, on *any* input string of length n.

To give an example, consider the Turing machine of Section 4.2.1 that decides, using one tape, the language consisting of all palindromes. The tape head of this Turing machine moves from the left to the right, then back to the left, then to the right again, back to the left, etc. Each time it reaches the leftmost or rightmost symbol, it deletes this symbol. The running time of this Turing machine, on any input string of length n, is

$$O(1 + 2 + 3 + \ldots + n) = O(n^2).$$

On the other hand, the running time of the Turing machine of Section 4.2.2, which also decides the palindromes, but using two tapes instead of just one, is $O(n)$.

In Section 4.4, we mentioned that all computation models listed there are equivalent, in the sense that if a language can be decided in one model, it can be decided in any of the other models. We just saw, however, that the language consisting of all palindromes allows a faster algorithm on a two-tape Turing machine than on one-tape Turing machines. (Even though we did not prove this, it is true that $\Omega(n^2)$ is a lower bound on the running time to decide palindromes on a one-tape Turing machine.) The following theorem can be proved.

Theorem 6.1.2 *Let A be a language (resp. let F be a function) that can be decided (resp. computed) in time T by an algorithm of type M. Then there is an algorithm of type N that decides A (resp. computes F) in time T', where*

M	N	T'
k-tape Turing machine	*one-tape Turing machine*	$O(T^2)$
one-tape Turing machine	*Java program*	$O(T^2)$
Java program	*k-tape Turing machine*	$O(T^4)$

6.2 The complexity class P

Definition 6.2.1 We say that algorithm M decides the language A (resp. computes the function F) in *polynomial time*, if there exists an integer $k \geq 1$, such that the running time of M is $O(n^k)$, for any input string of length n.

It follows from Theorem 6.1.2 that this notion of "polynomial time" does not depend on the model of computation:

Theorem 6.2.2 *Consider the models of computation "Java program", "k-tape Turing machine", and "one-tape Turing machine". If a language can be decided (resp. a function can be computed) in polynomial time in one of these models, then it can be decided (resp. computed) in polynomial time in all of these models.*

Because of this theorem, we can define the following two complexity classes:

$$\mathbf{P} := \{A : \text{ the language } A \text{ is decidable in polynomial time}\},$$

and

$$\mathbf{FP} := \{F : \text{ the function } F \text{ is computable in polynomial time}\}.$$

6.2.1 Some examples

Palindromes

Let Pal be the language

$$Pal := \{w \in \{a,b\}^* : \ w \text{ is a palindrome}\}.$$

We have seen that there exists a one-tape Turing machine that decides Pal in $O(n^2)$ time. Therefore, $Pal \in \mathbf{P}$.

Some functions in FP

The following functions are in the class **FP**:

- $F_1 : \mathbb{N}_0 \to \mathbb{N}_0$ defined by $F_1(x) := x + 1$,
- $F_2 : \mathbb{N}_0^2 \to \mathbb{N}_0$ defined by $F_2(x, y) := x + y$,
- $F_3 : \mathbb{N}_0^2 \to \mathbb{N}_0$ defined by $F_3(x, y) := xy$.

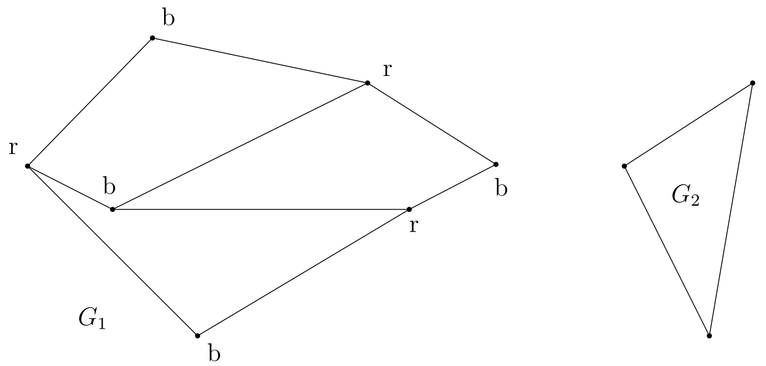


Figure 6.1: *The graph G_1 is 2-colorable;* r *stands for red;* b *stands for blue. The graph G_2 is not 2-colorable.*

Context-free languages

We have shown in Section 5.1.3 that every context-free language is decidable. The algorithm presented there, however, does not run in polynomial time. Using a technique called *dynamic programming* (which you will learn in COMP 3804), the following result can be shown:

Theorem 6.2.3 *Let Σ be an alphabet, and let $A \subseteq \Sigma^*$ be a context-free language. Then $A \in \mathbf{P}$.*

Observe that, obviously, every language in $\mathbf{P}$ is decidable.

The 2-coloring problem

Let G be a graph with vertex set V and edge set E. We say that G is *2-colorable*, if it is possible to give each vertex of V a color such that

1. for each edge $(u, v) \in E$, the vertices u and v have different colors, and
2. only two colors are used to color all vertices.

See Figure 6.1 for two examples. We define the following language:

$$2Color := \{\langle G \rangle : \text{ the graph } G \text{ is 2-colorable}\},$$

where $\langle G \rangle$ denotes the binary string that encodes the graph G.

We claim that $2\mathit{Color} \in \mathbf{P}$. In order to show this, we have to construct an algorithm that decides in polynomial time, whether or not any given graph is 2-colorable.

Let G be an arbitrary graph with vertex set $V = \{1, 2, \ldots, m\}$. The edge set of G is given by an *adjacency matrix*. This matrix, which we denote by E, is a two-dimensional array with m rows and m columns. For all i and j with $1 \leq i \leq m$ and $1 \leq j \leq m$, we have

$$E(i,j) = \begin{cases} 1 & \text{if } (i,j) \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

The length of the input G, i.e., the number of bits needed to specify G, is equal to $m^2 =: n$. We will present an algorithm that decides, in $O(n)$ time, whether or not the graph G is 2-colorable.

The algorithm uses the colors red and blue. It gives the first vertex the color red. Then, the algorithm considers all vertices that are connected by an edge to the first vertex, and colors them blue. Now the algorithm is done with the first vertex; it marks this first vertex.

Next, the algorithm chooses a vertex i that already has a color, but that has not been marked. Then it considers all vertices j that are connected by an edge to i. If j has the same color as i, then the input graph G is not 2-colorable. Otherwise, if vertex j does not have a color yet, the algorithm gives j the color that is different from i's color. After having done this for all neighbors j of i, the algorithm is done with vertex i, so it marks i.

It may happen that there is no vertex i that already has a color but that has not been marked. (In other words, each vertex i that is not marked does not have a color yet.) In this case, the algorithm chooses an arbitrary vertex i having this property, and colors it red. (This vertex i is the first vertex in its connected component that gets a color.)

This procedure is repeated until all vertices of G have been marked.

We now give a formal description of this algorithm. Vertex i has been *marked*, if

1. i has a color,
2. all vertices that are connected by an edge to i have a color, and
3. the algorithm has verified that each vertex that is connected by an edge to i has a color different from i's color.

The algorithm uses two arrays $f(1 \ldots m)$ and $a(1 \ldots m)$, and a variable M. The value of $f(i)$ is equal to the color (red or blue) of vertex i; if i does not have a color yet, then $f(i) = 0$. The value of $a(i)$ is equal to

$$a(i) = \begin{cases} 1 & \text{if vertex } i \text{ has been marked,} \\ 0 & \text{otherwise.} \end{cases}$$

The value of M is equal to the number of marked vertices. The algorithm is presented in Figure 6.2. You are encouraged to convince yourself of the correctness of this algorithm. That is, you should convince yourself that this algorithm returns YES if the graph G is 2-colorable, whereas it returns NO otherwise.

What is the running time of this algorithm? First we count the number of iterations of the outer while-loop. In one iteration, either M increases by one, or a vertex i, for which $a(i) = 0$, gets the color red. In the latter case, the variable M is increased during the next iteration of the outer while-loop. Since, during the entire outer while-loop, the value of M is increased from zero to m, it follows that there are at most $2m$ iterations of the outer while-loop. (In fact, the number of iterations is equal to m plus the number of connected components of G minus one.)

One iteration of the outer while-loop takes $O(m)$ time. Hence, the total running time of the algorithm is $O(m^2)$, which is $O(n)$. Therefore, we have shown that $2\mathit{Color} \in \mathbf{P}$.

6.3 The complexity class NP

Before we define the class **NP**, we consider some examples.

Example 6.3.1 Let G be a graph with vertex set V and edge set E, and let $k \geq 1$ be an integer. We say that G is *k-colorable*, if it is possible to give each vertex of V a color such that

1. for each edge $(u, v) \in E$, the vertices u and v have different colors, and
2. at most k different colors are used to color all vertices.

We define the following language:

$$k\mathit{Color} := \{\langle G \rangle : \text{ the graph } G \text{ is } k\text{-colorable}\}.$$

```
Algorithm 2COLOR
for i := 1 to m do f(i) := 0; a(i) := 0 endfor;
f(1) := red; M := 0;
while M ≠ m
do (* Find the minimum index i for which vertex i has not
      been marked, but has a color already *)
   bool := false; i := 1;
   while bool = false and i ≤ m
   do if a(i) = 0 and f(i) ≠ 0 then bool := true else i := i + 1 endif;
   endwhile;
   (* If bool = true, then i is the smallest index such that
      a(i) = 0 and f(i) ≠ 0.
      If bool = false, then for all i, the following holds: if a(i) = 0, then
      f(i) = 0; because M < m, there is at least one such i. *)
   if bool = true
   then for j := 1 to m
        do if E(i, j) = 1
           then if f(i) = f(j)
                then return NO and terminate
                else if f(j) = 0
                     then if f(i) = red
                          then f(j) := blue
                          else f(j) := red
                          endif
                     endif
                endif
           endif
        endfor;
        a(i) := 1; M := M + 1;
   else i := 1;
        while a(i) ≠ 0 do i := i + 1 endwhile;
        (* an unvisited connected component starts at vertex i *)
        f(i) := red
   endif
endwhile;
return YES
```

Figure 6.2: *An algorithm that decides whether or not a graph G is 2-colorable.*

We have seen that for $k = 2$, this problem is in the class **P**. For $k \geq 3$, it is not known whether there exists an algorithm that decides, in polynomial time, whether or not any given graph is k-colorable. In other words, for

$k \geq 3$, it is not known whether or not $k\mathit{Color}$ is in the class $\mathbf{P}$.

Example 6.3.2 Let G be a graph with vertex set $V = \{1, 2, \ldots, m\}$ and edge set E. A *Hamilton cycle* is a cycle in G that visits each vertex exactly once. Formally, it is a sequence $v_1, v_2, \ldots, v_m$ of vertices such that

1. $\{v_1, v_2, \ldots, v_m\} = V$, and

2. $\{(v_1, v_2), (v_2, v_3), \ldots, (v_{m-1}, v_m), (v_m, v_1)\} \subseteq E$.

We define the following language:

$$\mathit{HC} := \{\langle G \rangle : \text{ the graph } G \text{ contains a Hamilton cycle}\}.$$

It is not known whether or not HC is in the class $\mathbf{P}$.

Example 6.3.3 The *sum of subset* language is defined as follows:

$$\begin{aligned} \mathit{SOS} := \{\langle a_1, a_2, \ldots, a_m, b \rangle : \quad & m, a_1, a_2, \ldots, a_m, b \in \mathbb{N}_0 \text{ and} \\ & \exists I \subseteq \{1, 2, \ldots, m\}, \textstyle\sum_{i \in I} a_i = b\}. \end{aligned}$$

Also in this case, no polynomial-time algorithm is known that decides the language SOS. That is, it is not known whether or not SOS is in the class $\mathbf{P}$.

Example 6.3.4 An integer $x \geq 2$ is a prime number, if there are no $a, b \in \mathbb{N}$ such that $a \neq x$, $b \neq x$, and $x = ab$. Hence, the language of all non-primes that are greater than or equal to two, is

$$\mathit{NPrim} := \{\langle x \rangle : \; x \geq 2 \text{ and } x \text{ is not a prime number}\}.$$

It is not obvious at all, whether or not NPrim is in the class $\mathbf{P}$. In fact, it was shown only in 2002 that NPrim is in the class $\mathbf{P}$.

Observation 6.3.5 *The four languages above have the following in common: If someone gives us a "solution" for any given input, then we can easily, i.e., in polynomial time, verify whether or not this "solution" is a correct solution. Moreover, for any input to each of these four problems, there exists a "solution" whose length is polynomial in the length of the input.*

Let us again consider the language *kColor*. Let G be a graph with vertex set $V = \{1, 2, \ldots, m\}$ and edge set E, and let k be a positive integer. We want to decide whether or not G is k-colorable. A "solution" is a coloring of the nodes using at most k different colors. That is, a solution is a sequence $f_1, f_2, \ldots, f_m$. (Interpret this as: vertex i receives color f_i, $1 \leq i \leq m$). This sequence is a correct solution if and only if

1. $f_i \in \{1, 2, \ldots, k\}$, for all i with $1 \leq i \leq m$, and
2. for all i with $1 \leq i \leq m$, and for all j with $1 \leq j \leq m$, if $(i, j) \in E$, then $f_i \neq f_j$.

If someone gives us this solution (i.e., the sequence $f_1, f_2, \ldots, f_m$), then we can verify in polynomial time whether or not these two conditions are satisfied. The length of this solution is $O(m \log k)$: for each i, we need about $\log k$ bits to represent f_i. Hence, the length of the solution is polynomial in the length of the input, i.e., it is polynomial in the number of bits needed to represent the graph G and the number k.

For the Hamilton cycle problem, a solution consists of a sequence v_1, $v_2, \ldots, v_m$ of vertices. This sequence is a correct solution if and only if

1. $\{v_1, v_2, \ldots, v_m\} = \{1, 2, \ldots, m\}$ and
2. $\{(v_1, v_2), (v_2, v_3), \ldots, (v_{m-1}, v_m), (v_m, v_1)\} \subseteq E$.

These two conditions can be verified in polynomial time. Moreover, the length of the solution is polynomial in the length of the input graph.

Consider the sum of subset problem. A solution is a sequence $c_1, c_2, \ldots, c_m$. It is a correct solution if and only if

1. $c_i \in \{0, 1\}$, for all i with $1 \leq i \leq m$, and
2. $\sum_{i=1}^{m} c_i a_i = b$.

Hence, the set $I \subseteq \{1, 2, \ldots, m\}$ in the definition of *SOS* is the set of indices i for which $c_i = 1$. Again, these two conditions can be verified in polynomial time, and the length of the solution is polynomial in the length of the input.

Finally, let us consider the language *NPrim*. Let $x \geq 2$ be an integer. The integers a and b form a "solution" for x if and only if

1. $2 \leq a < x$,
2. $2 \leq b < x$, and
3. $x = ab$.

Clearly, these three conditions can be verified in polynomial time. Moreover, the length of this solution, i.e., the total number of bits in the binary representations of a and b, is polynomial in the number of bits in the binary representation of x.

Languages having the property that the correctness of a proposed "solution" can be verified in polynomial time, form the class **NP**:

Definition 6.3.6 A language A belongs to the class **NP**, if there exist a polynomial p and a language $B \in \mathbf{P}$, such that for every string w,

$$w \in A \Longleftrightarrow \exists s : |s| \leq p(|w|) \text{ and } \langle w, s \rangle \in B.$$

In words, a language A is in the class **NP**, if for every string w, $w \in A$ if and only if the following two conditions are satisfied:

1. There is a "solution" s, whose length $|s|$ is polynomial in the length of w (i.e., $|s| \leq p(|w|)$, where p is a polynomial).
2. In polynomial time, we can verify whether or not s is a correct "solution" for w (i.e., $\langle w, s \rangle \in B$ and $B \in \mathbf{P}$).

Hence, the language B can be regarded to be the "verification language":

$$B = \{\langle w, s \rangle : \; s \text{ is a correct "solution" for } w\}.$$

We have given already informal proofs of the fact that the languages *kColor*, *HC*, *SOS*, and *NPrim* are all contained in the class **NP**. Below, we formally prove that $NPrim \in \mathbf{NP}$. To prove this claim, we have to specify the polynomial p and the language $B \in \mathbf{P}$. First, we observe that

$$\begin{aligned} NPrim = \{\langle x \rangle : \quad & \text{there exist } a \text{ and } b \text{ in } \mathbb{N} \text{ such that} \\ & 2 \leq a < x, 2 \leq b < x \;\text{ and } x = ab \;\}. \end{aligned} \tag{6.1}$$

We define the polynomial p by $p(n) := n + 2$, and the language B as

$$B := \{\langle x, a, b \rangle : x \geq 2, 2 \leq a < x, 2 \leq b < x \text{ and } x = ab\}.$$

It is obvious that $B \in \mathbf{P}$: For any three positive integers x, a, and b, we can verify in polynomial time whether or not $\langle x, a, b\rangle \in B$. In order to do this, we only have to verify whether or not $x \geq 2$, $2 \leq a < x$, $2 \leq b < x$, and $x = ab$. If all these four conditions are satisfied, then $\langle x, a, b\rangle \in B$. If at least one of them is not satisfied, then $\langle x, a, b\rangle \notin B$.

It remains to show that for all $x \in \mathbb{N}$:

$$\langle x\rangle \in \mathit{NPrim} \Longleftrightarrow \exists a, b : |\langle a, b\rangle| \leq |\langle x\rangle| + 2 \text{ and } \langle x, a, b\rangle \in B. \tag{6.2}$$

(Remember that $|\langle x\rangle|$ denotes the number of bits in the binary representation of x; $|\langle a, b\rangle|$ denotes the total number of bits of a and b, i.e., $|\langle a, b\rangle| = |\langle a\rangle| + |\langle b\rangle|$.)

Let $x \in \mathit{NPrim}$. It follows from (6.1) that there exist a and b in $\mathbb{N}$, such that $2 \leq a < x$, $2 \leq b < x$, and $x = ab$. Since $x = ab \geq 2 \cdot 2 = 4 \geq 2$, it follows that $\langle x, a, b\rangle \in B$. Hence, it remains to show that

$$|\langle a, b\rangle| \leq |\langle x\rangle| + 2.$$

The binary representation of x contains $\lfloor \log x\rfloor + 1$ bits, i.e., $|\langle x\rangle| = \lfloor \log x\rfloor + 1$. We have

$$\begin{aligned}
|\langle a, b\rangle| &= |\langle a\rangle| + |\langle b\rangle| \\
&= (\lfloor \log a\rfloor + 1) + (\lfloor \log b\rfloor + 1) \\
&\leq \log a + \log b + 2 \\
&= \log ab + 2 \\
&= \log x + 2 \\
&\leq \lfloor \log x\rfloor + 3 \\
&= |\langle x\rangle| + 2.
\end{aligned}$$

This proves one direction of (6.2).

To prove the other direction, we assume that there are positive integers a and b, such that $|\langle a, b\rangle| \leq |\langle x\rangle| + 2$ and $\langle x, a, b\rangle \in B$. Then it follows immediately from (6.1) and the definition of the language B, that $x \in \mathit{NPrim}$. Hence, we have proved the other direction of (6.2). This completes the proof of the claim that

$$\mathit{NPrim} \in \mathbf{NP}.$$

6.3.1 P is contained in NP

Intuitively, it is clear that $\mathbf{P} \subseteq \mathbf{NP}$, because a language is

- in $\mathbf{P}$, if for every string w, it is possible to *compute* the "solution" s in polynomial time,
- in $\mathbf{NP}$, if for every string w and for any given "solution" s, it is possible to *verify* in polynomial time whether or not s is a correct solution for w (hence, we do not need to compute the solution s ourselves, we only have to verify it).

We give a formal proof of this:

Theorem 6.3.7 $\mathbf{P} \subseteq \mathbf{NP}$.

Proof. Let $A \in \mathbf{P}$. We will prove that $A \in \mathbf{NP}$. Define the polynomial p by $p(n) := 0$ for all $n \in \mathbb{N}_0$, and define

$$B := \{\langle w, \epsilon \rangle : w \in A\}.$$

Since $A \in \mathbf{P}$, the language B is also contained in $\mathbf{P}$. It is easy to see that

$$w \in A \Longleftrightarrow \exists s : |s| \leq p(|w|) = 0 \text{ and } \langle w, s \rangle \in B.$$

This completes the proof. ∎

6.3.2 Deciding NP-languages in exponential time

Let us look again at the definition of the class $\mathbf{NP}$. Let A be a language in this class. Then there exist a polynomial p and a language $B \in \mathbf{P}$, such that for all strings w,

$$w \in A \Longleftrightarrow \exists s : |s| \leq p(|w|) \text{ and } \langle w, s \rangle \in B. \tag{6.3}$$

How do we decide whether or not any given string w belongs to the language A? If we can find a string s that satisfies the right-hand side in (6.3), then we know that $w \in A$. On the other hand, if there is no such string s, then $w \notin A$. How much time do we need to decide whether or not such a string s exists?

Algorithm NonPrime
(∗ decides whether or not $\langle x \rangle \in NPrim$ ∗)
if $x = 0$ or $x = 1$ or $x = 2$
then return NO and terminate
else $a := 2$;
    while $a < x$
    do if $x \bmod a = 0$
        then return YES and terminate
        else $a := a + 1$
        endif
    endwhile;
    return NO
endif

Figure 6.3: *An algorithm that decides whether or not a number x is contained in the language NPrim.*

For example, let A be the language *NPrim*, and let $x \in \mathbb{N}$. The algorithm in Figure 6.3 decides whether or not $\langle x \rangle \in NPrim$.

It is clear that this algorithm is correct. Let n be the length of the binary representation of x, i.e., $n = \lfloor \log x \rfloor + 1$. If $x > 2$ and x is a prime number, then the while-loop makes $x-2$ iterations. Therefore, since $n-1 = \lfloor \log x \rfloor \leq \log x$, the running time of this algorithm is at least

$$x - 2 \geq 2^{n-1} - 2,$$

i.e., it is at least *exponential* in the length of the input.

We now prove that every language in **NP** can be decided in exponential time. Let A be an arbitrary language in **NP**. Let p be the polynomial, and let $B \in \mathbf{P}$ be the language such that for all strings w,

$$w \in A \Longleftrightarrow \exists s : |s| \leq p(|w|) \text{ and } \langle w, s \rangle \in B. \qquad (6.4)$$

The following algorithm decides, for any given string w, whether or not $w \in A$. It does so by looking at *all* possible strings s for which $|s| \leq p(|w|)$:

for all s with $|s| \leq p(|w|)$
do if $\langle w, s \rangle \in B$

then return YES and terminate
endif
endfor;
return NO

The correctness of the algorithm follows from (6.4). What is the running time? We assume that w and s are represented as binary strings. Let n be the length of the input, i.e., $n = |w|$.

How many binary strings s are there whose length is at most $p(|w|)$? Any such s can be described by a sequence of length $p(|w|) = p(n)$, consisting of the symbols "0", "1", and the blank symbol. Hence, there are at most $3^{p(n)}$ many binary strings s with $|s| \leq p(n)$. Therefore, the for-loop makes at most $3^{p(n)}$ iterations.

Since $B \in \mathbf{P}$, there is an algorithm and a polynomial q, such that this algorithm, when given any input string z, decides in $q(|z|)$ time, whether or not $z \in B$. This input z has the form $\langle w, s \rangle$, and we have

$$|z| = |w| + |s| \leq |w| + p(|w|) = n + p(n).$$

It follows that the total running time of our algorithm that decides whether or not $w \in A$, is bounded from above by

$$\begin{aligned} 3^{p(n)} \cdot q(n + p(n)) &\leq 2^{2p(n)} \cdot q(n + p(n)) \\ &\leq 2^{2p(n)} \cdot 2^{q(n+p(n))} \\ &= 2^{p'(n)}, \end{aligned}$$

where p' is the polynomial that is defined by $p'(n) := 2p(n) + q(n + p(n))$.

If we define the class **EXP** as

$$\mathbf{EXP} := \{A : \text{ there exists a polynomial } p \text{, such that } A \text{ can be decided in time } 2^{p(n)}\},$$

then we have proved the following theorem.

Theorem 6.3.8 **NP** $\subseteq$ **EXP**.

6.3.3 Summary

- **P** $\subseteq$ **NP**. It is not known whether **P** is a *proper* subclass of **NP**, or whether **P** = **NP**. This is one of the most important open problems in computer science. If you can solve this problem, then you will get one million dollars; not from us, but from the Clay Mathematics Institute, see

`http://www.claymath.org/prizeproblems/index.htm`

Most people believe that **P** is a proper subclass of **NP**.

- **NP** $\subseteq$ **EXP**, i.e., each language in **NP** can be decided in exponential time. It is not known whether **NP** is a proper subclass of **EXP**, or whether **NP** = **EXP**.

- It follows from **P** $\subseteq$ **NP** and **NP** $\subseteq$ **EXP**, that **P** $\subseteq$ **EXP**. It can be shown that **P** is a proper subset of **EXP**, i.e., there exist languages that can be decided in exponential time, but that cannot be decided in polynomial time.

- **P** is the class of those languages that can be decided *efficiently*, i.e., in polynomial time. Sets that are not in **P**, are not efficiently decidable.

6.4 Non-deterministic algorithms

The abbreviation **NP** stands for Non-deterministic Polynomial time. The algorithms that we have considered so far are *deterministic*, which means that at any time during the computation, the next computation step is uniquely determined. In a *non-deterministic* algorithm, there are one or more possibilities for being the next computation step, and the algorithm chooses one of them.

To give an example, we consider the language *SOS*, see Example 6.3.3. Let m, a_1, a_2, $\ldots$, a_m, and b be elements of $\mathbb{N}_0$. Then

$$\langle a_1, a_2, \ldots, a_m, b\rangle \in SOS \iff \text{there exist } c_1, c_2, \ldots, c_m \in \{0,1\}, \text{ such that } \textstyle\sum_{i=1}^{m} c_i a_i = b.$$

The following non-deterministic algorithm decides the language *SOS*:

Algorithm SOS$(m, a_1, a_2, \ldots, a_m, b)$:
$s := 0$;
for $i := 1$ **to** m
do $s := s \mid s := s + a_i$

endfor;
if $s = b$
then return YES
else return NO
endif

The line

$$s := s \mid s := s + a_i$$

means that either the instruction "$s := s$" or the instruction "$s := s + a_i$" is executed.

Let us assume that $\langle a_1, a_2, \ldots, a_m, b\rangle \in SOS$. Then there are $c_1, c_2, \ldots, c_m \in \{0,1\}$ such that $\sum_{i=1}^m c_i a_i = b$. Assume our algorithm does the following, for each i with $1 \leq i \leq m$: In the i-th iteration,

- if $c_i = 0$, then it executes the instruction "$s := s$",
- if $c_i = 1$, then it executes the instruction "$s := s + a_i$".

Then after the for-loop, we have $s = b$, and the algorithm returns YES; hence, the algorithm has correctly found out that $\langle a_1, a_2, \ldots, a_m, b\rangle \in SOS$. In other words, in this case, *there exists at least one accepting computation.*

On the other hand, if $\langle a_1, a_2, \ldots, a_m, b\rangle \notin SOS$, then the algorithm always returns NO, no matter which of the two instructions is executed in each iteration of the for-loop. In this case, there is *no accepting computation.*

Definition 6.4.1 Let M be a non-deterministic algorithm. We say that M *accepts* a string w, if there exists at least one computation that, on input w, returns YES.

Definition 6.4.2 We say that a non-deterministic algorithm M *decides* a language A *in time* T, if for every string w, the following holds: $w \in A$ if and only if there exists at least one computation that, on input w, returns YES and that takes at most $T(|w|)$ time.

The non-deterministic algorithm that we have seen above decides the language SOS in linear time: Let $\langle a_1, a_2, \ldots, a_m, b\rangle \in SOS$, and let n be the length of this input. Then

$$n = |\langle a_1\rangle| + |\langle a_2\rangle| + \ldots + |\langle a_m\rangle| + |\langle b\rangle| \geq m.$$

For this input, there is a computation that returns YES and that takes $O(m) = O(n)$ time.

As in Section 6.2, we define the notion of "polynomial time" for non-deterministic algorithms. The following theorem relates this notion to the class **NP** that we defined in Definition 6.3.6.

Theorem 6.4.3 *A language A is in the class* **NP** *if and only if there exists a non-deterministic Turing machine (or Java program) that decides A in polynomial time.*

6.5 NP-complete languages

Languages in the class **P** are considered *easy*, i.e., they can be decided in polynomial time. People believe (but cannot prove) that **P** is a proper subclass of **NP**. If this is true, then there are languages in **NP** that are *hard*, i.e., cannot be decided in polynomial time.

Intuition tells us that if $\mathbf{P} \neq \mathbf{NP}$, then the *hardest* languages in **NP** are not contained in **P**. These languages are called **NP**-*complete*. In this section, we will give a formal definition of this concept.

If we want to talk about the "hardest" languages in **NP**, then we have to be able to compare two languages according to their "difficulty". The idea is as follows: We say that a language B is "at least as hard" as a language A, if the following holds: If B can be decided in polynomial time, then A can also be decided in polynomial time.

Definition 6.5.1 Let $A \subseteq \{0,1\}^*$ and $B \subseteq \{0,1\}^*$ be languages. We say that $A \leq_P B$, if there exists a function

$$f : \{0,1\}^* \to \{0,1\}^*$$

such that

1. $f \in \mathbf{FP}$ and
2. for all strings w in $\{0,1\}^*$,

$$w \in A \Longleftrightarrow f(w) \in B.$$

If $A \leq_P B$, then we also say that "B is at least as hard as A", or "A is polynomial-time reducible to B".

We first show that this formal definition is in accordance with the intuitive definition given above.

Theorem 6.5.2 *Let A and B be languages such that $B \in \mathbf{P}$ and $A \leq_P B$. Then $A \in \mathbf{P}$.*

Proof. Let $f : \{0,1\}^* \rightarrow \{0,1\}^*$ be the function in **FP** for which

$$w \in A \Longleftrightarrow f(w) \in B. \tag{6.5}$$

The following algorithm decides whether or not any given binary string w is in A:

> $u := f(w)$;
> **if** $u \in B$
> **then** return YES
> **else** return NO
> **endif**

The correctness of this algorithm follows immediately from (6.5). So it remains to show that the running time is polynomial in the length of the input string w.

Since $f \in \mathbf{FP}$, there exists a polynomial p such that the function f can be computed in time p. Similarly, since $B \in \mathbf{P}$, there exists a polynomial q, such that the language B can be decided in time q.

Let n be the length of the input string w, i.e., $n = |w|$. Then the length of the string u is less than or equal to $p(|w|) = p(n)$. (Why?) Therefore, the running time of our algorithm is bounded from above by

$$p(|w|) + q(|u|) \leq p(n) + q(p(n)).$$

Since the function p', defined by $p'(n) := p(n) + q(p(n))$, is a polynomial, this proves that $A \in \mathbf{P}$. ■

The following theorem states that the relation $\leq_P$ is reflexive and transitive. We leave the proof as an exercise.

Theorem 6.5.3 *Let A, B, and C be languages. Then*

1. *$A \leq_P A$, and*

2. *if $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.*

We next show that the languages in **P** are the *easiest* languages in **NP**:

Theorem 6.5.4 *Let A be a language in* **P**, *and let B be an arbitrary language such that $B \neq \emptyset$ and $B \neq \{0,1\}^*$. Then $A \leq_P B$.*

Proof. We choose two strings u and v in $\{0,1\}^*$, such that $u \in B$ and $v \notin B$. (Observe that this is possible.) Define the function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ by

$$f(w) := \begin{cases} u & \text{if } w \in A, \\ v & \text{if } w \notin A. \end{cases}$$

Then it is clear that for any binary string w,

$$w \in A \Longleftrightarrow f(w) \in B.$$

Since $A \in \mathbf{P}$, the function f can be computed in polynomial time, i.e., $f \in \mathbf{FP}$. ∎

6.5.1 Two examples of reductions

Sum of subsets and knapsacks

We start with a simple reduction. Consider the two languages

$$\begin{aligned} SOS := \{\langle a_1, \ldots, a_m, b\rangle : \quad & m, a_1, \ldots, a_m, b \in \mathbb{N}_0 \text{ and there exist} \\ & c_1, \ldots, c_m \in \{0,1\}, \text{ such that } \textstyle\sum_{i=1}^m c_i a_i = b\} \end{aligned}$$

and

$$\begin{aligned} KS \quad := \quad & \{\langle w_1, \ldots, w_m, k_1, \ldots, k_m, W, K\rangle : \\ & m, w_1, \ldots, w_m, k_1, \ldots, k_m, W, K \in \mathbb{N}_0 \\ & \text{and there exist } c_1, \ldots, c_m \in \{0,1\}, \\ & \text{such that } \textstyle\sum_{i=1}^m c_i w_i \leq W \text{ and } \sum_{i=1}^m c_i k_i \geq K\}. \end{aligned}$$

The notation *KS* stands for *knapsack*: We have m pieces of food. The i-th piece has weight w_i and contains k_i calories. We want to decide whether or not we can fill our knapsack with a subset of the pieces of food such that the total weight is at most W, and the total amount of calories is at least K.

Theorem 6.5.5 *$SOS \leq_P KS$.*

Proof. Let us first see what we have to show. According to Definition 6.5.1, we need a function $f \in \mathbf{FP}$, that maps input strings for *SOS* to input strings for *KS*, in such a way that

$$\langle a_1, \ldots, a_m, b \rangle \in SOS \Longleftrightarrow f(\langle a_1, \ldots, a_m, b \rangle) \in KS.$$

In order for $f(\langle a_1, \ldots, a_m, b \rangle)$ to be an input string for *KS*, this function value has to be of the form

$$f(\langle a_1, \ldots, a_m, b \rangle) = \langle w_1, \ldots, w_m, k_1, \ldots, k_m, W, K \rangle.$$

We define

$$f(\langle a_1, \ldots, a_m, b \rangle) := \langle a_1, \ldots, a_m, a_1, \ldots, a_m, b, b \rangle.$$

It is clear that $f \in \mathbf{FP}$. We have

$$\begin{aligned}
&\langle a_1, \ldots, a_m, b \rangle \in SOS \\
&\Longleftrightarrow \text{ there exist } c_1, \ldots, c_m \in \{0, 1\} \text{ such that } \textstyle\sum_{i=1}^m c_i a_i = b \\
&\Longleftrightarrow \text{ there exist } c_1, \ldots, c_m \in \{0, 1\} \text{ such that } \textstyle\sum_{i=1}^m c_i a_i \leq b \text{ and } \sum_{i=1}^m c_i a_i \geq b \\
&\Longleftrightarrow \langle a_1, \ldots, a_m, a_1, \ldots, a_m, b, b \rangle \in KS \\
&\Longleftrightarrow f(\langle a_1, \ldots, a_m, b \rangle) \in KS.
\end{aligned}$$

∎

Cliques and Boolean formulas

We will define two languages $A = 3SAT$ and $B = Clique$ that have, at first sight, nothing to do with each other. Then we show that, nevertheless, $A \leq_P B$.

Let G be a graph with vertex set V and edge set E. A subset V' of V is called a *clique*, if each pair of distinct vertices in V' is connected by an edge in E. We define the following language:

$$Clique := \{\langle G, k \rangle : \ k \in \mathbb{N} \text{ and } G \text{ has a clique with } k \text{ vertices}\}.$$

We encourage you to prove the following claim:

Theorem 6.5.6 *Clique* $\in$ **NP**.

Next we consider *Boolean formulas* φ, with variables $x_1, x_2, \ldots, x_m$, having the form

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k, \tag{6.6}$$

where each C_i, $1 \leq i \leq k$, is of the form

$$C_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i.$$

Each ℓ_a^i is either a variable or the negation of a variable. An example of such a formula is

$$\varphi = (x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4).$$

A formula φ of the form (6.6) is said to be *satisfiable*, if there exists a truth-value in $\{0, 1\}$ for each of the variables $x_1, x_2, \ldots, x_m$, such that the entire formula φ is true. Our example formula is satisfiable: If we take $x_1 = 0$ and $x_2 = 1$, and give x_3 and x_4 an arbitrary value, then

$$\varphi = (0 \vee 1 \vee 0) \wedge (x_3 \vee 1 \vee x_4) \wedge (1 \vee \neg x_3 \vee \neg x_4) = 1.$$

We define the following language:

$$3SAT := \{\langle \varphi \rangle : \varphi \text{ is of the form (6.6) and is satisfiable}\}.$$

Again, we encourage you to prove the following claim:

Theorem 6.5.7 $3SAT \in$ **NP**.

Observe that the elements of *Clique* (which are pairs consisting of a graph and a positive integer) are completely different from the elements of $3SAT$ (which are Boolean formulas). We will show that $3SAT \leq_P$ *Clique*. Recall that this means the following: If the language *Clique* can be decided in polynomial time, then the language $3SAT$ can also be decided in polynomial time. In other words, any polynomial-time algorithm that decides *Clique* can be converted to a polynomial-time algorithm that decides $3SAT$.

Theorem 6.5.8 $3SAT \leq_P$ *Clique*.

Proof. We have to show that there exists a function $f \in \mathbf{FP}$, that maps input strings for $3SAT$ to input strings for *Clique*, such that for each Boolean formula φ that is of the form (6.6),

$$\langle \varphi \rangle \in 3SAT \Longleftrightarrow f(\langle \varphi \rangle) \in \mathit{Clique}.$$

The function f maps the binary string encoding an arbitrary Boolean formula φ to a binary string encoding a pair (G, k), where G is a graph and k is a positive integer. We have to define this function f in such a way that

$$\varphi \text{ is satisfiable} \Longleftrightarrow G \text{ has a clique with } k \text{ vertices.}$$

Let

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$$

be an arbitrary Boolean formula in the variables $x_1, x_2, \ldots, x_m$, where each C_i, $1 \leq i \leq k$, is of the form

$$C_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i.$$

Remember that each ℓ_a^i is either a variable or the negation of a variable.

The formula φ is mapped to the pair (G, k), where the vertex set V and the edge set E of the graph G are defined as follows:

- $V = \{v_1^1, v_2^1, v_3^1, \ldots, v_1^k, v_2^k, v_3^k\}$. The idea is that each vertex v_a^i corresponds to one term ℓ_a^i.
- The pair (v_a^i, v_b^j) of vertices form an edge in E if and only if
  - $i \neq j$ and
  - ℓ_a^i is not the negation of ℓ_b^j.

To give an example, let φ be the Boolean formula

$$\varphi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3), \tag{6.7}$$

i.e., $k = 3$, $C_1 = x_1 \vee \neg x_2 \vee \neg x_3$, $C_2 = \neg x_1 \vee x_2 \vee x_3$, and $C_3 = x_1 \vee x_2 \vee x_3$. The graph G that corresponds to this formula is given in Figure 6.4.

It is not difficult to see that the function f can be computed in polynomial time. So it remains to prove that

$$\varphi \text{ is satisfiable} \Longleftrightarrow G \text{ has a clique with } k \text{ vertices.} \tag{6.8}$$

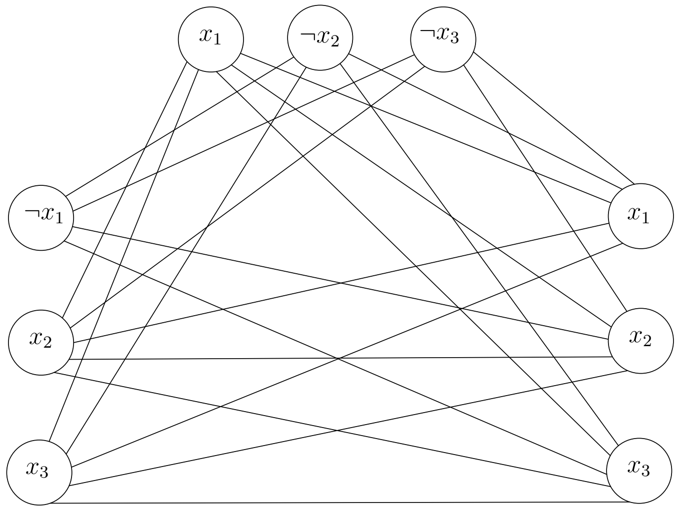


Figure 6.4: *The formula φ in (6.7) is mapped to this graph. The vertices on the top represent C_1; the vertices on the left represent C_2; the vertices on the right represent C_3.*

To prove this, we first assume that the formula

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$$

is satisfiable. Then there exists a truth-value in $\{0, 1\}$ for each of the variables $x_1, x_2, \ldots, x_m$, such that the entire formula φ is true. Hence, for each i with $1 \leq i \leq k$, there is at least one term ℓ_a^i in

$$C_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i$$

that is true (i.e., has value 1).

Let V' be the set of vertices obtained by choosing for each i, $1 \leq i \leq k$, exactly one vertex v_a^i such that ℓ_a^i has value 1.

It is clear that V' contains exactly k vertices. We claim that this set is a clique in G. To prove this claim, let v_a^i and v_b^j be two distinct vertices in V'. It follows from the definition of V' that $i \neq j$ and $\ell_a^i = \ell_b^j = 1$. Hence, ℓ_a^i is not the negation of ℓ_b^j. But this means that the vertices v_a^i and v_b^j are connected by an edge in G.

This proves one direction of (6.8). To prove the other direction, we assume that the graph G contains a clique V' with k vertices.

The vertices of G consist of k groups, where each group contains exactly three vertices. Since vertices within the same group are not connected by edges, the clique V' contains exactly one vertex from each group. Hence, for each i with $1 \leq i \leq k$, there is exactly one a, such that $v_a^i \in V'$. Consider the corresponding term ℓ_a^i. We know that this term is either a variable or the negation of a variable, i.e., ℓ_a^i is either of the form x_j or of the form $\neg x_j$. If $\ell_a^i = x_j$, then we give x_j the truth-value 1. Otherwise, we have $\ell_a^i = \neg x_j$, in which case we give x_j the truth-value 0. Since V' is a clique, each variable gets at most one truth-value. If a variable has no truth-value yet, then we give it an arbitrary truth-value.

If we substitute these truth-values into φ, then the entire formula has value 1. Hence, φ is satisfiable. ∎

In order to get a better understanding of this proof, you should verify the proof for the formula φ in (6.7) and the graph G in Figure 6.4.

6.5.2 Definition of NP-completeness

Reductions, as defined in Definition 6.5.1, allow us to compare two language according to their difficulty. A language B in **NP** is called **NP**-complete, if B belongs to the *most difficult* languages in **NP**; in other words, B is at least as hard as *any* other language in **NP**.

Definition 6.5.9 Let $B \subseteq \{0,1\}^*$ be a language. We say that B is **NP**-*complete*, if

1. $B \in \mathbf{NP}$ and
2. $A \leq_P B$, for *every* language A in **NP**.

Theorem 6.5.10 *Let B be an* **NP**-*complete language. Then*

$$B \in \mathbf{P} \Longleftrightarrow \mathbf{P} = \mathbf{NP}.$$

Proof. Intuitively, this theorem should be true: If the language B is in **P**, then B is an easy language. On the other hand, since B is **NP**-complete, it belongs to the most difficult languages in **NP**. Hence, the most difficult language in **NP** is easy. But then all languages in **NP** must be easy, i.e., $\mathbf{P} = \mathbf{NP}$.

We give a formal proof. Let us first assume that $B \in \mathbf{P}$. We already know that $\mathbf{P} \subseteq \mathbf{NP}$. Hence, it remains to show that $\mathbf{NP} \subseteq \mathbf{P}$. Let A be an arbitrary language in **NP**. Since B is **NP**-complete, we have $A \leq_P B$. Then, by Theorem 6.5.2, we have $A \in \mathbf{P}$.

To prove the converse, assume that $\mathbf{P} = \mathbf{NP}$. Since $B \in \mathbf{NP}$, it follows immediately that $B \in \mathbf{P}$. ∎

Theorem 6.5.11 *Let B and C be languages, such that $C \in \mathbf{NP}$ and $B \leq_P C$. If B is* **NP***-complete, then C is also* **NP***-complete.*

Proof. First, we give an intuitive explanation of the claim: By assumption, B belongs to the most difficult languages in **NP**, and C is at least as hard as B. Since $C \in \mathbf{NP}$, it follows that C belongs to the most difficult languages in **NP**. Hence, C is **NP**-complete.

To give a formal proof, we have to show that $A \leq_P C$, for all languages A in **NP**. Let A be an arbitrary language in **NP**. Since B is **NP**-complete, we have $A \leq_P B$. Since $B \leq_P C$, it follows from Theorem 6.5.3, that $A \leq_P C$. Therefore, C is **NP**-complete. ∎

Theorem 6.5.11 can be used to prove the **NP**-completeness of languages: Let C be a language, and assume that we want to prove that C is **NP**-complete. We can do this in the following way:

1. We first prove that $C \in \mathbf{NP}$.
2. Then we find a language B that looks "similar" to C, and for which we already know that it is **NP**-complete.
3. Finally, we prove that $B \leq_P C$.
4. Then, Theorem 6.5.11 tells us that C is **NP**-complete.

Of course, this leads to the question "How do we know that the language B is **NP**-complete?" In order to apply Theorem 6.5.11, we need a "first" **NP**-complete language; the **NP**-completeness of this language must be proven using Definition 6.5.9.

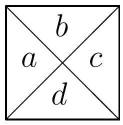
Observe that it is not clear at all that there exist **NP**-complete languages! For example, consider the language $3SAT$. If we want to use Definition 6.5.9 to show that this language is **NP**-complete, then we have to show that

- $3SAT \in \mathbf{NP}$. We know from Theorem 6.5.7 that this is true.
- $A \le_P 3SAT$, for *every* language $A \in \mathbf{NP}$. Hence, we have to show this for languages A such as $kColor$, HC, SOS, $NPrim$, KS, $Clique$, and for *infinitely* many other languages.

In 1971, Cook has exactly done this: He showed that the language $3SAT$ is **NP**-complete. Since his proof is rather technical, we will prove the **NP**-completeness of another language.

6.5.3 An NP-complete domino game

We are given a finite collection of *tile types*. For each such type, there are arbitrarily many tiles of this type. A *tile* is a square that is partitioned into four triangles. Each of these triangles contains a symbol that belongs to a finite alphabet Σ. Hence, a tile looks as follows:



We are also given a square *frame*, consisting of cells. Each cell has the same size as a tile, and contains a symbol of Σ.

The problem is to decide whether or not this *domino game* has a solution. That is, can we completely fill the frame with tiles such that

- for any two neighboring tiles s and s', the two triangles of s and s' that touch each other contain the same symbol, and
- each triangle that touches the frame contains the same symbol as the cell of the frame that is touched by this triangle.

There is one final restriction: The orientation of the tiles is fixed, they cannot be rotated.

Let us give a formal definition of this problem. We assume that the symbols belong to the finite alphabet $\Sigma = \{0,1\}^m$, i.e., each symbol is encoded as a bit-string of length m. Then, a tile type can be encoded as a tuple of four bit-strings, i.e., as an element of Σ^4. A frame consisting of t rows and t columns can be encoded as a string in Σ^{4t}.

We denote the language of all solvable domino games by *Domino*:

$$\begin{aligned} Domino \; := \; & \{\langle m, k, t, R, T_1, \dots, T_k\rangle : \\ & m \geq 1, k \geq 1, t \geq 1, R \in \Sigma^{4t}, T_i \in \Sigma^4, 1 \leq i \leq k, \\ & \text{frame } R \text{ can be filled using tiles of types} \\ & T_1, \dots, T_k.\} \end{aligned}$$

We will prove the following theorem.

Theorem 6.5.12 *The language Domino is* **NP***-complete.*

Proof. It is clear that $Domino \in \mathbf{NP}$: A solution consists of a $t \times t$ matrix, in which the (i, j)-entry indicates the type of the tile that occupies position (i, j) in the frame. The number of bits needed to specify such a solution is polynomial in the length of the input. Moreover, we can verify in polynomial time whether or not any given "solution" is correct.

It remains to show that

$$A \leq_P Domino, \text{ for every language } A \text{ in } \mathbf{NP}.$$

Let A be an arbitrary language in **NP**. Then there exist a polynomial p and a non-deterministic Turing machine M, that decides the language A in time p. We may assume that this Turing machine has only one tape.

On input $w = a_1 a_2 \dots a_n$, the Turing machine M starts in the start state z_0, with its tape head on the cell containing the symbol a_1. We may assume that during the entire computation, the tape head never moves to the left of this initial cell. Hence, the entire computation "takes place" in and to the right of the initial cell. We know that

$$\begin{aligned} w \in A \quad \Longleftrightarrow \quad & \text{on input } w, \text{ there exists an accepting computation} \\ & \text{that makes at most } p(n) \text{ computation steps.} \end{aligned}$$

At the end of such an accepting computation, the tape only contains the symbol 1, which we may assume to be in the initial cell, and M is in the final state z_1. In this case, we may assume that the accepting computation makes exactly $p(n)$ computation steps. (If this is not the case, then we extend the computation using the instruction $z_1 1 \rightarrow z_1 1 N$.)

We need one more technical detail: We may assume that $za \rightarrow z'bR$ and $za' \rightarrow z''b'L$ are not both instructions of M. Hence, the state of the Turing machine uniquely determines the direction in which the tape head moves.

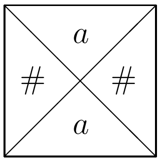
We have to define a domino game, that depends on the input string w and the Turing machine M, such that

$$w \in A \Longleftrightarrow \text{this domino game is solvable.}$$

The idea is to encode an accepting computation of the Turing machine M as a solution of the domino game. In order to do this, we use a frame in which each row corresponds to one computation step. This frame consists of $p(n)$ rows. Since an accepting computation makes exactly $p(n)$ computation steps, and since the tape head never moves to the left of the initial cell, this tape head can visit only $p(n)$ cells. Therefore, our frame will have $p(n)$ columns.

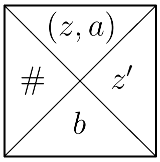
The domino game will use the following tile types:

1. For each symbol a in the alphabet of the Turing machine M:



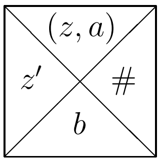
   Intuition: Before and after the computation step, the tape head is not on this cell.

2. For each instruction $za \to z'bR$ of the Turing machine M:



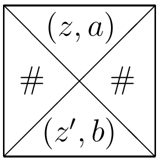
   Intuition: Before the computation step, the tape head is on this cell; the tape head makes one step to the right.

3. For each instruction $za \to z'bL$ of the Turing machine M:



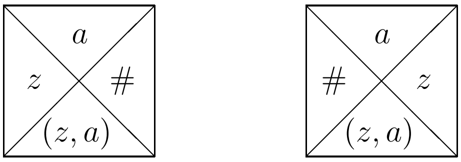
Intuition: Before the computation step, the tape head is on this cell; the tape head makes one step to the left.

4. For each instruction $za \to z'bN$ of the Turing machine M:



Intuition: Before and after the computation step, the tape head is on this cell.

5. For each state z and for each symbol a in the alphabet of the Turing machine M, there are two tile types:



Intuition: The leftmost tile indicates that the tape head enters this cell from the left; the righmost tile indicates that the tape head enters this cell from the right.

This specifies all tile types. The $p(n) \times p(n)$ frame is given in Figure 6.5. The top row corresponds to the start of the computation, whereas the bottom row corresponds to the end of the computation. The left and right columns correspond to the part of the tape in which the tape head can move.

The encodings of these tile types and the frame can be computed in polynomial time.

It can be shown that, for any input string w, any accepting computation of length $p(n)$ of the Turing machine M can be encoded as a solution of this domino game. Conversely, any solution of this domino game can be "translated" to an accepting computation of length $p(n)$ of M, on input string w. Hence, the following holds.

$$\begin{aligned} w \in A &\iff \text{there exists an accepting computation that makes} \\ &\ \ p(n) \text{ computation steps} \\ &\iff \text{the domino game is solvable.} \end{aligned}$$

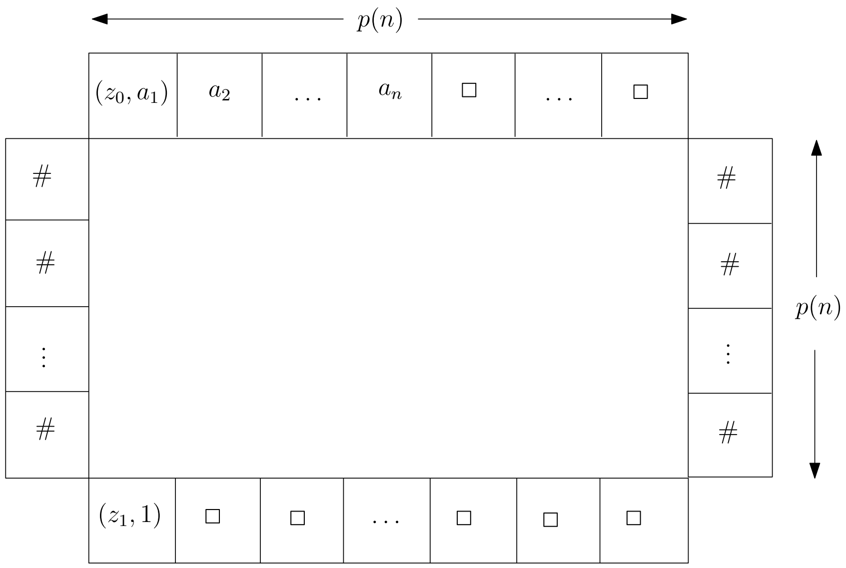


Figure 6.5: *The $p(n) \times p(n)$ frame for the domino game.*

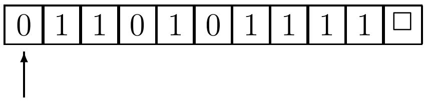
Therefore, we have $A \leq_P$ *Domino*. Hence, the language *Domino* is **NP**-complete. ∎

An example of a domino game

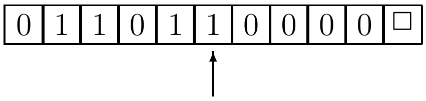
We have defined the domino game corresponding to a Turing machine that solves a decision problem. Of course, we can also do this for Turing machines that compute functions. In this section, we will exactly do this for a Turing machine that computes the successor function $x \rightarrow x + 1$.

We will design a Turing machine with one tape, that gets as input the binary representation of a natural number x, and that computes the binary representation of $x + 1$.

Start of the computation: The tape contains a 0 followed by the binary representation of the integer $x \in \mathbb{N}_0$. The tape head is on the leftmost bit (which is 0), and the Turing machine is in the start state z_0. Here is an example, where $x = 431$:



End of the computation: The tape contains the binary representation of the number $x + 1$. The tape head is on the rightmost 1, and the Turing machine is in the final state z_1. For our example, the tape looks as follows:



Our Turing machine will use the following states:

z_0 : start state; tape head moves to the right
z_1 : final state
z_2 : tape head moves to the left; on its way to the left, it has not read 0

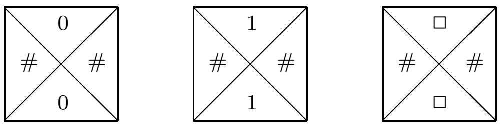
The Turing machine has the following instructions:

$$
\begin{array}{ll}
z_0 0 \rightarrow z_0 0 R & \quad z_2 1 \rightarrow z_2 0 L \\
z_0 1 \rightarrow z_0 1 R & \quad z_2 0 \rightarrow z_1 1 N \\
z_0 \Box \rightarrow z_2 \Box L &
\end{array}
$$

In Figure 6.6, you can see the sequence of states and tape contents of this Turing machine on input $x = 11$.

We now construct the domino game that corresponds to the computation of this Turing machine on input $x = 11$. Following the general construction in Section 6.5.3, we obtain the following tile types:

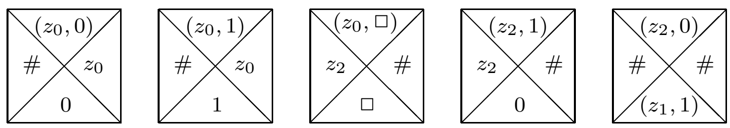
1. The three symbols of the alphabet yield three tile types:



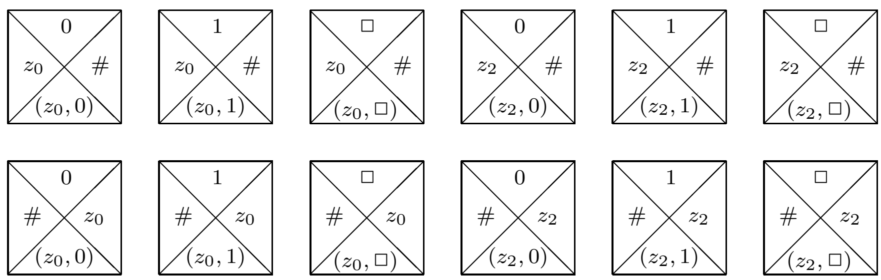
2. The five instructions of the Turing machine yield five tile types:

$(z_0, 0)$	1	0	1	1	$\Box$
0	$(z_0, 1)$	0	1	1	$\Box$
0	1	$(z_0, 0)$	1	1	$\Box$
0	1	0	$(z_0, 1)$	1	$\Box$
0	1	0	1	$(z_0, 1)$	$\Box$
0	1	0	1	1	$(z_0, \Box)$
0	1	0	1	$(z_2, 1)$	$\Box$
0	1	0	$(z_2, 1)$	0	$\Box$
0	1	$(z_2, 0)$	0	0	$\Box$
0	1	$(z_1, 1)$	0	0	$\Box$

Figure 6.6: *The computation of the Turing machine on input $x = 11$. The pair (state,symbol) indicates the position of the tape head.*



3. The states z_0 and z_2, and the three symbols of the alphabet yield twelve tile types:



The computation of the Turing machine on input $x = 11$ consists of nine computation steps. During this computation, the tape head visits exactly six cells. Therefore, the frame for the domino game has nine rows and six columns. This frame is given in Figure 6.7. In Figure 6.8, you find the solution of the domino game. Observe that this solution is nothing but an equivalent way of writing the computation of Figure 6.6. Hence, the computation of the Turing machine corresponds to a solution of the domino game; in fact, the converse also holds.

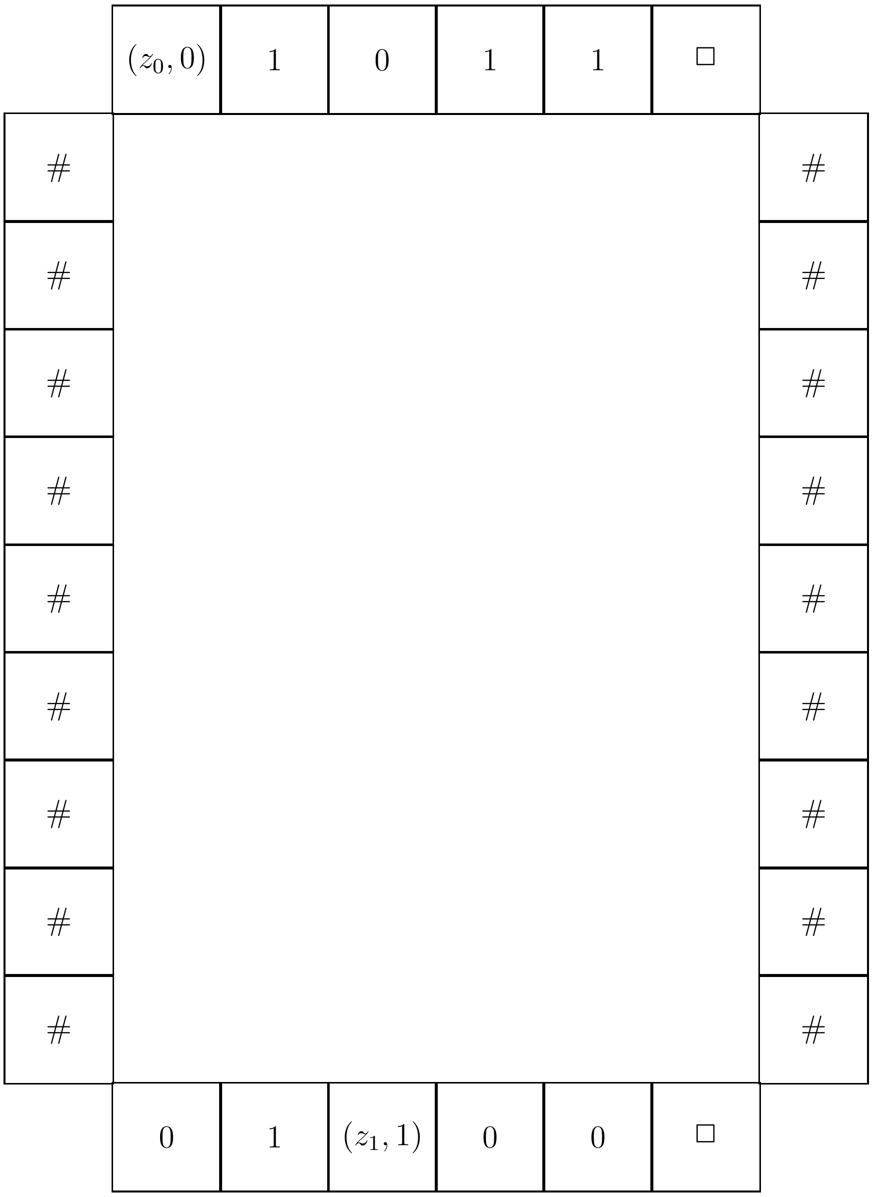


Figure 6.7: *The frame for the domino game for input $x = 11$.*

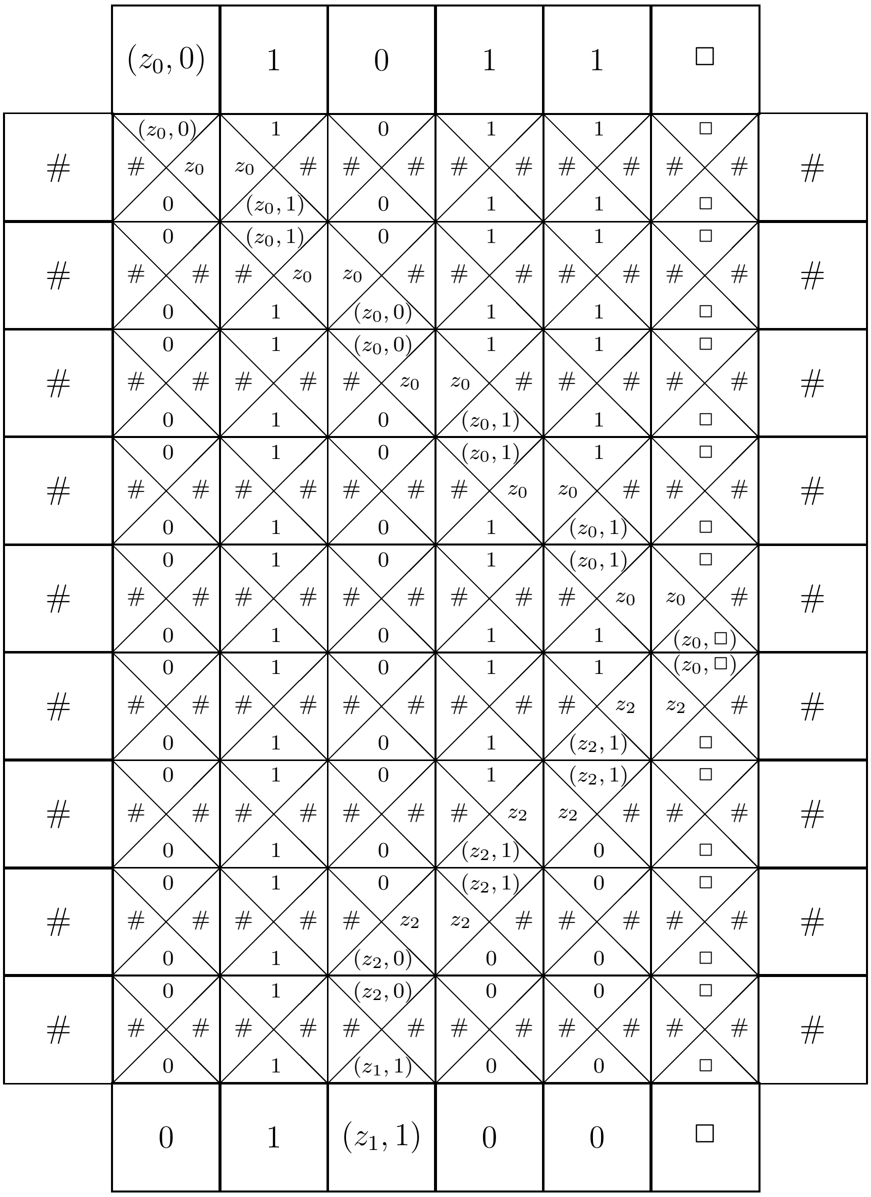


Figure 6.8: *The solution for the domino game for input $x = 11$.*

6.5.4 Examples of NP-complete languages

In Section 6.5.3, we have shown that *Domino* is **NP**-complete. Using this result, we will apply Theorem 6.5.11 to prove the **NP**-completeness of some other languages.

Satisfiability

We consider Boolean formulas φ, in the variables $x_1, x_2, \ldots, x_m$, having the form

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k, \tag{6.9}$$

where each C_i, $1 \leq i \leq k$, is of the following form:

$$C_i = \ell_1^i \vee \ell_2^i \vee \ldots \vee \ell_{k_i}^i.$$

Each ℓ_j^i is either a variable or the negation of a variable. Such a formula φ is said to be *satisfiable*, if there exists a truth-value in $\{0,1\}$ for each of the variables $x_1, x_2, \ldots, x_m$, such that the entire formula φ is true. We define the following language:

$$SAT := \{\langle\varphi\rangle : \varphi \text{ is of the form (6.9) and is satisfiable}\}.$$

We will prove that SAT is **NP**-complete.

It is clear that $SAT \in$ **NP**. If we can show that

$$Domino \leq_P SAT,$$

then it follows from Theorem 6.5.11 that SAT is **NP**-complete. (In Theorem 6.5.11, take $B :=$ *Domino* and $C := SAT$.)

Hence, we need a function $f \in$ **FP**, that maps input strings for *Domino* to input strings for SAT, in such a way that for every domino game D, the following holds:

$$\text{domino game } D \text{ is solvable} \Longleftrightarrow \begin{array}{l}\text{the formula encoded by the}\\ \text{string } f(\langle D\rangle) \text{ is satisfiable.}\end{array} \tag{6.10}$$

Let us consider an arbitrary domino game D. Let k be the number of tile types, and let the frame have t rows and t columns. We denote the tile types by $T_1, T_2, \ldots, T_k$.

We map this domino game D to a Boolean formula φ, such that (6.10) holds. The formula φ will have variables

$$x_{ij\ell}, 1 \leq i \leq t, 1 \leq j \leq t, 1 \leq \ell \leq k.$$

These variables can be interpretated as follows:

$$x_{ij\ell} = 1 \Longleftrightarrow \text{there is a tile of type } T_\ell \text{ at position } (i,j) \text{ of the frame.}$$

We define:

- For all i and j with $1 \leq i \leq t$ and $1 \leq j \leq t$:

  $$C^1_{ij} := x_{ij1} \vee x_{ij2} \vee \ldots \vee x_{ijk}.$$

  This formula expresses the condition that there is at least one tile at position (i,j).

- For all i, j, ℓ and ℓ' with $1 \leq i \leq t$, $1 \leq j \leq t$, and $1 \leq \ell < \ell' \leq k$:

  $$C^2_{ij\ell\ell'} := \neg x_{ij\ell} \vee \neg x_{ij\ell'}.$$

  This formula expresses the condition that there is at most one tile at position (i,j).

- For all i, j, ℓ and ℓ' with $1 \leq i \leq t$, $1 \leq j < t$, $1 \leq \ell \leq k$ and $1 \leq \ell' \leq k$, such that $i < t$ and the right symbol on a tile of type T_ℓ is not equal to the left symbol on a tile of type $T_{\ell'}$:

  $$C^3_{ij\ell\ell'} := \neg x_{ij\ell} \vee \neg x_{i,j+1,\ell'}.$$

  This formula expresses the condition that neighboring tiles in the same row "fit" together. There are symmetric formulas for neighboring tiles in the same column.

- For all j and ℓ with $1 \leq j \leq t$ and $1 \leq \ell \leq k$, such that the top symbol on a tile of type T_ℓ is not equal to the symbol at position j of the upper boundary of the frame:

  $$C^4_{j\ell} := \neg x_{1j\ell}.$$

  This formula expresses the condition that tiles that touch the upper boundary of the frame "fit" there. There are symmetric formulas for the lower, left, and right boundaries of the frame.

The formula φ is the conjunction of all these formulas C^1_{ij}, $C^2_{ij\ell\ell'}$, $C^3_{ij\ell\ell'}$, and $C^4_{j\ell}$. The complete formula φ consists of

$$O(t^2k + t^2k^2 + t^2k^2 + tk) = O(t^2k^2)$$

terms, i.e., its length is polynomial in the length of the domino game. This implies that φ can be constructed in polynomial time. Hence, the function f that maps the domino game D to the Boolean formula φ, is in the class **FP**. It is not difficult to see that (6.10) holds for this function f. Therefore, we have proved the following result.

Theorem 6.5.13 *The language SAT is* **NP***-complete.*

In Section 6.5.1, we have defined the language $3SAT$.

Theorem 6.5.14 *The language $3SAT$ is* **NP***-complete.*

Proof. It is clear that $3SAT \in$ **NP**. If we can show that

$$SAT \leq_P 3SAT,$$

then the claim follows from Theorem 6.5.11. Let

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$$

be an input for SAT, in the variables $x_1, x_2, \ldots, x_m$. We map φ, in polynomial time, to an input φ' for $3SAT$, such that

$$\varphi \text{ is satisfiable} \Longleftrightarrow \varphi' \text{ is satisfiable.} \tag{6.11}$$

For each i with $1 \leq i \leq k$, we do the following. Consider

$$C_i = \ell^i_1 \vee \ell^i_2 \vee \ldots \vee \ell^i_{k_i}.$$

- If $k_i = 1$, then we define

$$C'_i := \ell^i_1 \vee \ell^i_1 \vee \ell^i_1.$$

- If $k_i = 2$, then we define

$$C'_i := \ell^i_1 \vee \ell^i_2 \vee \ell^i_2.$$

- If $k_i = 3$, then we define
$$C'_i := C_i.$$
- If $k_i \geq 4$, then we define
$$\begin{aligned} C'_i \;\; := \;\; & (\ell^i_1 \vee \ell^i_2 \vee z^i_1) \wedge (\neg z^i_1 \vee \ell^i_3 \vee z^i_2) \wedge (\neg z^i_2 \vee \ell^i_4 \vee z^i_3) \wedge \ldots \\ & \wedge (\neg z^i_{k_i-3} \vee \ell^i_{k_i-1} \vee \ell^i_{k_i}), \end{aligned}$$
  where $z^i_1, \ldots, z^i_{k_i-3}$ are new variables.

Let
$$\varphi' := C'_1 \wedge C'_2 \wedge \ldots \wedge C'_k.$$
Then φ' is an input for $3SAT$, and (6.11) holds. ∎

Theorems 6.5.6, 6.5.8, 6.5.11, and 6.5.14 imply:

Theorem 6.5.15 *The language Clique is* **NP***-complete.*

The traveling salesperson problem

We are given two positive integers k and m, a set of m cities, and an integer $m \times m$ matrix M, where
$$M(i,j) = \text{the cost of driving from city } i \text{ to city } j,$$
for all $i, j \in \{1, 2, \ldots, m\}$. We want to decide whether or not there is a tour through all cities whose total cost is less than or equal to k. This problem is **NP**-complete.

Bin packing

We are given three positive integers m, k, and ℓ, a set of m objects having volumes $a_1, a_2, \ldots, a_m$, and k bins. Each bin has volume ℓ. We want to decide whether or not all objects fit within these bins. This problem is **NP**-complete.

Here is another interpretation of this problem: We are given m jobs that need time $a_1, a_2, \ldots, a_m$ to complete. We are also given k processors, and an integer ℓ. We want to decide whether or not it is possible to divide the jobs over the k processors, such that no processor needs more than ℓ time.

Time tables

We are given a set of courses, class rooms, and professors. We want to decide whether or not there exists a time table such that all courses are being taught, no two courses are taught at the same time in the same class room, no professor teaches two courses at the same time, and conditions such as "Prof. L. Azy does not teach before 1pm" are satisfied. This problem is **NP**-complete.

Motion planning

We are given two positive integers k and ℓ, a set of k polyhedra, and two points s and t in $\mathbb{Q}^3$. We want to decide whether or not there exists a path between s and t, that does not intersect any of the polyhedra, and whose length is less than or equal to ℓ. This problem is **NP**-complete.

Map labeling

We are given a map with m cities, where each city is represented by a point. For each city, we are given a rectangle that is large enough to contain the name of the city. We want to decide whether or not these rectangles can be placed on the map, such that

- no two rectangles overlap,
- For each i with $1 \leq i \leq m$, the point that represents city i is a corner of its rectangle.

This problem is **NP**-complete.

This list of **NP**-complete problems can be extended almost arbitrarily: For thousands of problems, it is known that they are **NP**-complete. For all of these, it is *not known*, whether or not they can be solved efficiently (i.e., in polynomial time). Collections of **NP**-complete problems can be found in the book

- M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of* **NP**-*Completeness.* W.H. Freeman, New York, 1979,

and on the web page

`http://www.nada.kth.se/~viggo/wwwcompendium/`

Exercises

6.1 Prove that the function $F : \mathbb{N} \to \mathbb{N}$, defined by $F(x) := 2^x$, is not in **FP**.

6.2 Prove Theorem 6.5.3.

6.3 Prove that the language *Clique* is in the class **NP**.

6.4 Prove that the language $3SAT$ is in the class **NP**.

6.5 We define the following languages:

- Sum of subset:

$$SOS := \{\langle a_1, a_2, \ldots, a_m, b\rangle : \exists I \subseteq \{1, 2, \ldots, m\}, \sum_{i \in I} a_i = b\}.$$

- Set partition:

$$SP := \{\langle a_1, a_2, \ldots, a_m\rangle : \exists I \subseteq \{1, 2, \ldots, m\}, \sum_{i \in I} a_i = \sum_{i \notin I} a_i\}.$$

- Bin packing: BP is the set of all strings $\langle s_1, s_2, \ldots, s_m, B\rangle$ for which

  1. $0 < s_i < 1$, for all i,
  2. $B \in \mathbb{N}$,
  3. the numbers $s_1, s_2, \ldots, s_m$ fit into B bins, where each bin has size one, i.e., there exists a partition of $\{1, 2, \ldots, m\}$ into subsets I_k, $1 \leq k \leq B$, such that $\sum_{i \in I_k} s_i \leq 1$ for all k, $1 \leq k \leq B$.

  For example, $\langle 1/6, 1/2, 1/5, 1/9, 3/5, 1/5, 1/2, 11/18, 3\rangle \in BP$, because the eight fractions fit into three bins:

$$1/6 + 1/9 + 11/18 \leq 1, \ 1/2 + 1/2 = 1, \ \text{and } 1/5 + 3/5 + 1/5 = 1.$$

1. Prove that $SOS \leq_P SP$.
2. Prove that the language SOS is **NP**-complete. You may use the fact that the language SP is **NP**-complete.

3. Prove that the language BP is **NP**-complete. Again, you may use the fact that the language SP is **NP**-complete.

6.6 Prove that $3Color \leq_P 3SAT$.

Hint: For each vertex i, and for each of the three colors k, introduce a Boolean variable x_{ik}.

6.7 The $(0,1)$-*integer programming* language IP is defined as follows:

$$\begin{aligned} IP := \{\langle A, c\rangle : \quad & A \text{ is an integer } m \times n \text{ matrix for some } m, n \in \mathbb{N}, \\ & c \text{ is an integer vector of length } m, \text{ and} \\ & \exists x \in \{0,1\}^n \text{ such that } Ax \leq c \text{ (componentwise) } \}. \end{aligned}$$

Prove that the language IP is **NP**-complete. You may use the fact that the language SOS is **NP**-complete.

6.8 Let φ be a Boolean formula in the variables $x_1, x_2, \ldots, x_m$.

We say that φ is in *disjunctive normal form* (DNF) if it is of the form

$$\varphi = C_1 \vee C_2 \vee \ldots \vee C_k, \tag{6.12}$$

where each C_i, $1 \leq i \leq k$, is of the following form:

$$C_i = \ell_1^i \wedge \ell_2^i \wedge \ldots \wedge \ell_{k_i}^i.$$

Each ℓ_j^i is a *literal*, which is either a variable or the negation of a variable.

We say that φ is in *conjunctive normal form* (CNF) if it is of the form

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k, \tag{6.13}$$

where each C_i, $1 \leq i \leq k$, is of the following form:

$$C_i = \ell_1^i \vee \ell_2^i \vee \ldots \vee \ell_{k_i}^i.$$

Again, each ℓ_j^i is a literal.

We define the following two languages:

$$DNFSAT := \{\langle\varphi\rangle : \varphi \text{ is in DNF-form and is satisfiable}\},$$

and

$$CNFSAT := \{\langle\varphi\rangle : \varphi \text{ is in CNF-form and is satisfiable}\}.$$

1. Prove that the language *DNFSAT* is in **P**.

2. What is wrong with the following argument: Since we can rewrite any Boolean formula in DNF-form, we have $CNFSAT \leq_P DNFSAT$. Hence, since *CNFSAT* is **NP**-complete and since $DNFSAT \in \mathbf{P}$, we have $\mathbf{P} = \mathbf{NP}$.

3. Prove directly that for every language A in **P**, $A \leq_P CNFSAT$. "Directly" means that you should not use the fact that *CNFSAT* is **NP**-complete.

6.9 Prove that the polynomial upper bound on the length of the string y in the definition of **NP** is necessary, in the sense that if it is left out, then any decidable language would satisfy the condition.

More precisely, we say that the language A belongs to the class **D**, if there exists a language $B \in \mathbf{P}$, such that for every string w,

$$w \in A \Longleftrightarrow \exists y : \langle w, y \rangle \in B.$$

Prove that **D** is equal to the class of all decidable languages.

Chapter 7

Summary

We have seen several different models for "processing" languages, i.e., processing sets of strings over some finite alphabet. For each of these models, we have asked the question which types of languages can be processed, and which types of languages cannot be processed. In this final chapter, we give a brief summary of these results.

Regular languages: This class of languages was considered in Chapter 2. The following statements are equivalent:

1. The language A is regular, i.e., there exists a deterministic finite automaton that accepts A.

2. There exists a nondeterministic finite automaton that accepts A.

3. There exists a regular expression that describes A.

This claim was proved by the following conversions:

1. Every nondeterministic finite automaton can be converted to an equivalent deterministic finite automaton.

2. Every deterministic finite automaton can be converted to an equivalent regular expression.

3. Every regular expression can be converted to an equivalent nondeterministic finite automaton.

We have seen that the class of regular languages is closed under the regular operations: If A and B are regular languages, then

1. $A \cup B$ is regular,
2. AB is regular,
3. A^* is regular,
4. $\overline{A}$ is regular, and
5. $A \cap B$ is regular.

Finally, the pumping lemma for regular languages gives a property that every regular language possesses. We have used this to prove that languages such as $\{a^n b^n : n \geq 0\}$ are not regular.

Context-free languages: This class of languages was considered in Chapter 3. We have seen that every regular language is context-free. Moreover, there exist languages, for example $\{a^n b^n : n \geq 0\}$, that are context-free, but not regular. The following statements are equivalent:

1. The language A is context-free, i.e., there exists a context-free grammar whose language is A.
2. There exists a context-free grammar in Chomsky normal form whose language is A.
3. There exists a nondeterministic pushdown automaton that accepts A.

This claim was proved by the following conversions:

1. Every context-free grammar can be converted to an equivalent context-free grammar in Chomsky normal form.
2. Every context-free grammar in Chomsky normal form can be converted to an equivalent nondeterministic pushdown automaton.
3. Every nondeterministic pushdown automaton can be converted to an equivalent context-free grammar. (This conversion was not covered in this book.)

Nondeterministic pushdown automata are more powerful than deterministic pushdown automata: There exists a nondeterministic pushdown automaton that accepts the language

$$\{vbw : v \in \{a,b\}^*, w \in \{a,b\}^*, |v| = |w|\},$$

but there is no deterministic pushdown automaton that accepts this language. (We did not prove this in this book.)

We have seen that the class of context-free languages is closed under the union, concatenation, and star operations: If A and B are context-free languages, then

1. $A \cup B$ is context-free,

2. AB is context-free, and

3. A^* is context-free.

However,

1. the intersection of two context-free languages is not necessarily context-free, and

2. the complement of a context-free language is not necessarily context-free.

Finally, the pumping lemma for context-free languages gives a property that every context-free language possesses. We have used this to prove that languages such as $\{a^n b^n c^n : n \geq 0\}$ are not context-free.

The Church-Turing Thesis: In Chapter 4, we considered "reasonable" computational devices that model real computers. Examples of such devices are Turing machines (with one or more tapes) and Java programs. It turns out that all known "reasonable" devices are equivalent, i.e., can be converted to each other. This led to the Church-Turing Thesis:

- Every computational process that is intuitively considered to be an algorithm can be converted to a Turing machine.

Decidable and enumerable languages: These classes of languages were considered in Chapter 5. They are defined based on "reasonable" computational devices, such as Turing machines and Java programs. We have seen that

1. every context-free language is decidable, and
2. every decidable language is enumerable.

Moreover,

1. there exist languages, for example $\{a^n b^n c^n : n \geq 0\}$, that are decidable, but not context-free,
2. there exist languages, for example the Halting Problem, that are enumerable, but not decidable,
3. there exist languages, for example the complement of the Halting Problem, that are not enumerable.

In fact,

1. the class of all languages is not countable, whereas
2. the class of all enumerable languages is countable.

The following statements are equivalent:

1. The language A is decidable.
2. Both A and its complement $\overline{A}$ are enumerable.

Complexity classes: These classes of languages were considered in Chapter 6.

1. The class **P** consists of all languages that can be decided in polynomial time by a deterministic Turing machine.
2. The class **NP** consists of all languages that can be decided in polynomial time by a nondeterministic Turing machine. Equivalently, a language A is in the class **NP**, if for every string $w \in A$, there exists a "solution" s, such that (i) the length of s is polynomial in the length of w, and (ii) the correctness of s can be verified in polynomial time.

The following properties hold:

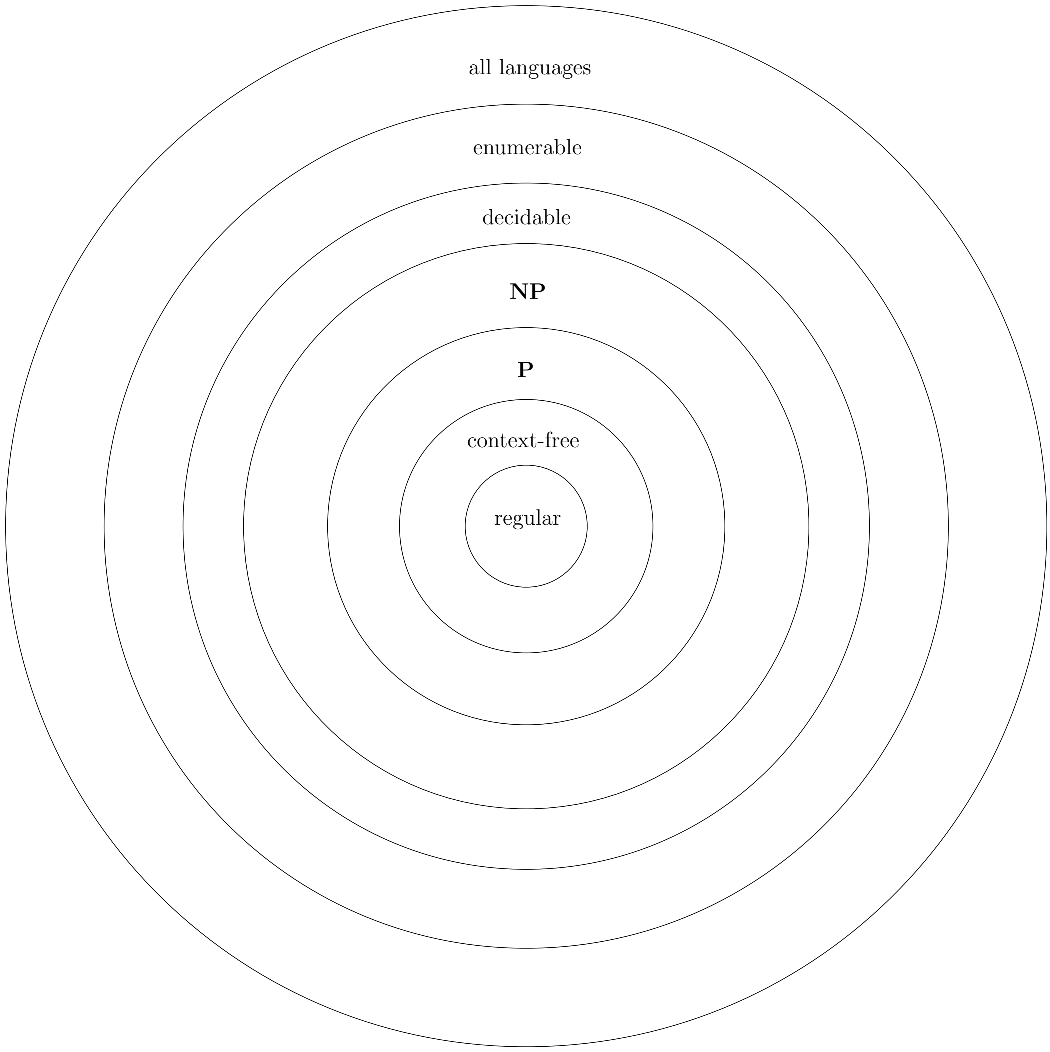
1. Every context-free language is in **P**. (We did not prove this).
2. Every language in **P** is also in **NP**.
3. It is not known if there exist languages that are in **NP**, but not in **P**.
4. Every language in **NP** is decidable.

We have introduced reductions to define the notion of a language B to be "at least as hard" as a language A. A language B is called **NP**-complete, if

1. B belongs to the class **NP**, and
2. B is at least as hard as every language in the class **NP**.

We have seen that **NP**-complete exist.

The figure below summarizes the relationships among the various classes of languages.


all languages
enumerable
decidable
NP
P
context-free
regular