Computability and Complexity

Computability and Complexity

Hubie Chen

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Times New Roman by Hubert Chen. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Chen, Hubie, author.
Title: Computability and complexity / Hubie Chen.
Description: Cambridge, Massachusetts : The MIT Press, [2023] | Includes bibliographical references and index.
Identifiers: LCCN 2022054401 | ISBN 9780262048620 (hardcover) | ISBN 9780262376860 (epub) | ISBN 9780262376853 (pdf)
Subjects: LCSH: Computational complexity.
Classification: LCC QA267.7 .C44 2023 | DDC 511.3/52–dc23/eng20230429
LC record available at https://lccn.loc.gov/2022054401

10 9 8 7 6 5 4 3 2 1

Contents

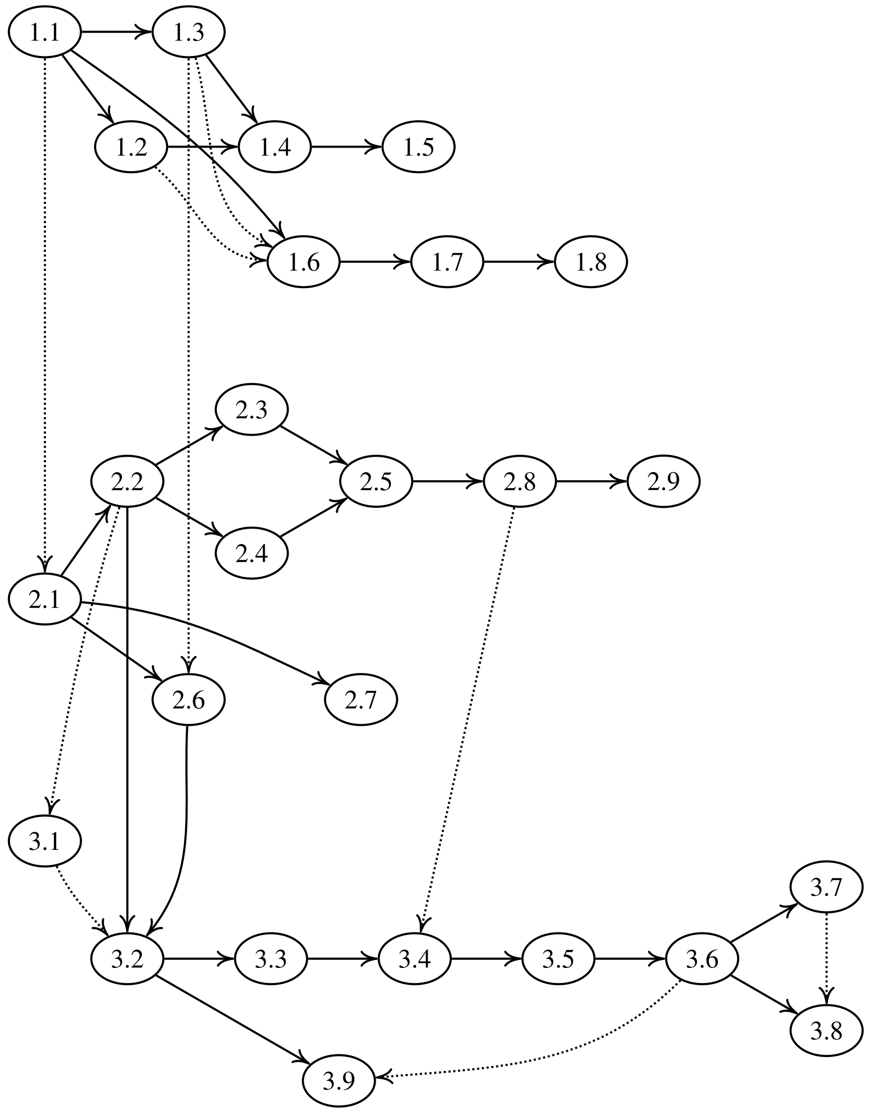


Figure 0.0.1. The dependencies between the sections of Chapters 1, 2, and 3. Each solid arrow indicates a strong dependency; each dotted arrow indicates a weak dependency.

Preface

This textbook is an introduction to the *theory of computation*, viewed here as the study of forms of computation that are abstract in the sense of being defined mathematically, and hence amenable to a mathematical treatment. These forms include general computation as typically associated with the term *algorithm*, time-efficient computation, and space-bounded computation. A key aim of this theory is to understand both the capabilities and limitations of each of these forms of computation; in part, this aim is achieved by comparing the forms to each other.

Audiences

This book is targeted to multiple audiences.

First, this book aspires to be useable in a computer science curriculum at the upper undergraduate level, and above. In particular, it was designed to be accessible to computer science undergraduates having a basic mathematical maturity—namely, comfort working with mathematical notation, definitions, and proofs.

At the same time, this book aims to serve as a thorough, rigorous initiation into the theory of computation which may be used by students, researchers, and workers in disciplines that draw on or depend upon this theory. This initiation should provide its users with the ability to begin engaging with research literature that employs the theory of computation, and a point of departure for learning more about this theory. This book could serve as a primary text or as a reference for both undergraduate-level and graduate-level courses that cover or contact the theory of computation. For all use cases, the crucial background is the aforementioned basic mathematical maturity.

This book's presentation assumes familiarity with basic set-theoretic notions (such as those of *set, subset, power set, intersection,* and *union*), functions, and propositional logic. On the part of the reader, some acquaintance with graph theory and with computer programming would be helpful, but is not strictly required.

Approach

This book attempts to offer a uniform treatment of core concepts and topics in the theory of computation. Throughout, an effort is made to underscore the unity of the subject and its methods of inquiry. A small number of recurrent themes are emphasized: computational models and the comparison of their respective language classes; closure properties of language classes; how determinism and nondeterminism compare in various contexts; and, notions of *reduction* and *completeness*. The treatment strives to sculpt these themes and the covered material into a coherent storyline in the hope of imbuing the reader with a sense of the beauty and mystery held by the subject. Indeed, the desire to maintain an overall narrative influenced not only the approach to the material, but also, to some extent, the choice of which results and topics to include.

Where relevant, the treatment points out alternative ways to approach the material. In addition, numerous remarks, notes, and exercises anticipate and explore ideas that deepen the main presentation. These features were included with the goals of imparting a rounded, robust appreciation of the subject, and of laying a foundation that naturally encourages and leads to further study.

In writing this book, I (the author!) endeavored to provide rigorous proofs of all major results. In fact, there are at least a few points in the book where, in lieu of waving my hands or requesting exercises from the reader, I elected to give arguments in significant detail. I did this with the philosophy that such detailed arguments should, at the minimum, be available to a reader wishing to see them, and with the understanding that a given reader should feel comfortable in skipping such arguments, especially upon initial readings. I have attempted to signal such detailed arguments. (Examples of proofs where such detail occurs include those of Theorem 1.4.1, Theorem 1.4.2, and Theorem 3.6.15; in these cases, I attempted to structure each proof so that the most detail-intensive portions occur in the latter part of the proof.)

Contents and use

A primary axis along which this book is organized is a presented procession of *computational models*, which are mathematical descriptions of computing devices. Following a time-honored tradition, the book begins by considering relatively restricted models called *finite-state automata*, which can process their inputs only by reading and with a finite amount of memory. These models are the subjects of Chapter 1; although restricted, they are well motivated and appealing in their own right, and provide meaningful preparation for the subsequent development. Chapter 2, on *computability theory*, introduces and studies the *Turing machine* model as a formalization of the notion of *algorithm*, and as representative of a fully fledged computational model. *Complexity theory*, the study of resource-bounded computation, is covered in Chapters 3 and 4. Chapter 3 studies time-bounded computation; *polynomial-time deterministic computation* is presented as a formalization of efficient computation, and the framework of *NP-completeness* complements it by offering an avenue for showing negative results. Chapter 4 presents a selection of further topics from complexity theory: space-bounded computation, hierarchy theorems, and parameterized complexity theory.

In writing this book, efforts were made to minimize dependencies between sections, so as to promote modularity and allow for multiple pathways through the material. Some suggestions as to how this book could be used in courses are as follows:

- A course covering automata, computability, and complexity could cover Sections 1.1–1.4 and 1.6; Sections 2.1–2.6 and 2.8; and, Sections 3.1–3.6, along with a selection of the reductions in Sections 3.8. Other sections could be added in optionally.
- A course focused on computability and complexity could cover Sections 2.1–2.8, Sections 3.1–3.7, and a selection of the reductions in Section 3.8. Other sections could be added in optionally.
- A course focused on complexity could cover Sections 2.1, 2.2, and 2.6; all Sections of Chapter 3, with a selection of reductions made from Section 3.8; and, a selection of topics from Chapter 4.

The beginning of Section 3.8 contains guidance on how one might form a selection of reductions, from this section, for study: see Remark 3.8.2.

The dependencies between the sections of Chapters 1 through 3 were shown in Figure 0.0.1 (a few pages ago). Let us describe the ways in which each section of Chapter 4 depends on prior sections. Section 4.1 depends on the same sections as Section 3.2 does—namely, Sections 2.1, 2.2, and 2.6; acquaintance with Section 3.2 is also useful for studying Section 4.1. Section 4.2 depends on Sections 3.2 and 4.1, and also, in a basic way, on Section 2.3. Sections 4.3 and 4.4 are intended to be read in sequence; they mainly depend on Section 3.2, although a general acquaintance with NP-completeness as presented in Chapter 3 is helpful. Section 4.5 depends on Section 3.2, and expects general knowledge of NP-completeness; familiarity with Section 4.3 is also of aid.

For the most part, I believe that this book's mathematical conventions are fairly standard. But there is one deviation from the norm that I wish to directly address here. In this book's treatment of complexity theory, where it would usually be said that a language is *in* $\mathcal{P}$, where $\mathcal{P}$ denotes the class of polynomial-time computable languages, I say adjectivally that the language *is PTIME* or that it is a *PTIME language*. I similarly state that a language *is NP* or *is coNP* where the norm would be to say that the language is *in* $\mathcal{NP}$ or *in* $\mathit{co}\mathcal{NP}$, respectively; here, $\mathcal{NP}$ and $\mathit{co}\mathcal{NP}$ denote the suggested complexity classes. My personal experience from teaching this book's material is that each instance of non-uniformity in presentation forms a potential stumbling block and a potential source of confusion for the student. In my view, the use of class notation in complexity theory forms such a stumbling block, as it is not at all standard to use class notation in automata theory or in computability theory: the tradition is to say adjectivally, for example, that a language *is regular*, as opposed to saying that it *is in* the class of regular languages; indeed, it is atypical to introduce a notation for denoting the class of regular languages. In favor of uniformity and consistency, I opted to maintain the adjectival approach in the treatment of complexity theory. I elected use of the adjective *PTIME* over the shorter alternative *P* due to its higher readability and higher descriptiveness. Despite these adjustments to the norm, due to the prevalence of class notation in complexity theory at large, I often show how statements can be alternatively presented using class notation where it is natural to do so.

Acknowledgments

For their feedback of many forms, I thank Eric Allender, Ilario Bonacina, Ronald de Haan, Montserrat Hermo, Neil Immerman, Bart M. P. Jansen, Jari J. H. de Kroon, Victor Lagerkvist, Benoit Larose, Moritz Müller, George Osipov, Riccardo Pucella, Friedrich Slivovsky, Johan Thapper, and Harry Vinall-Smeeth. I extend special thanks to Moritz Müller for aiding with a wide range of queries, and for discussions on how to approach a number of the covered topics. Curt Alexander, Christine Cuoco, and Joe Halpern provided useful advice for which I am grateful. I thank my editor Elizabeth Swayze for all of her patience and kind help.

I thank all of my teachers, in general; of all of them, I'll explicitly name my doctoral advisor, Dexter Kozen, to whom I'm grateful for sharing with me an abundance of mathematics, computer science, and ideas about exposition.

I thank my mother and my father for their continued support. For their generous and warm hospitality during a crucial stage of writing, I express gratitude to my in-laws Harumi-san, Tsuneo-san, Mika-san, Michio-san, and their family; I'll always fondly remember all of the time that we spent together. I am indebted to my mother-in-law Mariko for her extensive and sustained help during the final years of this project. For their companionship over many of this project's varied phases, I thank my wife Mayumi for her humor and backing, and for extending her decision-making capacities; and I thank my son Noah for all of his instinct, laughter, and curiosity. I thank my daughter Arisa for joining us in this world with a signature energy after this book's final draft was submitted, and for not waiting very long to begin smiling. Ari-chan, Noah-kun, and Mayumi, this book is dedicated to you.

Hubie Chen
London, 2022

Introduction

Just as the natural sciences aim to uncover, abstract out, and understand fundamental laws of nature, the *theory of computation* aims to distill and analyze basic principles governing computational phenomena—in particular, to understand both the capabilities and limitations of computation. This book strives to provide a solid grounding in the core concepts of this theory as it has been developed thus far.

This book's material is motivated by the following two focal questions:

- *What is computable?*
- *What is efficiently computable?*

We will interpret, approach, and answer these questions mathematically. In doing so, we will engage with a beautiful and intricate tapestry of ideas and concepts, which, we will argue, are of a timeless, indelible character. However, in order to initiate our acquaintance with this tapestry, these questions need to be made precise and we need to elucidate a couple of points.

First, we need to qualify the *what* in these questions, by specifying which *things* we will classify as being *computable* or not, and as being *efficiently computable* or not. *Languages* are the objects that we will focus on classifying in this way, where a language is a set of *strings*—fortunately, we will be able to give formal definitions of these notions relatively readily. A language can be alternatively viewed as a so-called *decision problem*, which provides an infinitude of yes-or-no questions: given as input a string x, decide whether or not x belongs to the language. As we will see, the definition of language is sufficiently generic that we will be able to take various sets of objects and encode them as languages—for example, sets of graphs, or sets of natural numbers.

Next, we need to define what it means for a language to be *computable* or *efficiently computable*. Robust definitions of these notions emerged in the first half and second half of the twentieth century, respectively. Presenting these definitions requires some development: to arrive at them, we will present and study so-called *computational models* (also known as *models of computation*), which are abstract, mathematically defined models of computing devices. For example, the first and simplest computational model that we will encounter is the *deterministic finite automaton (DFA)*. Each DFA M renders a judgement of *acceptance* or *rejection* on every input string, and thus has an associated language: its set of accepted strings, denoted by $L(M)$. A language is defined as *regular* when there exists a DFA M such that it is equal to $L(M)$. In this way, the DFA model gives rise to and defines the class of regular languages, the simplest language class with which we will engage. By enriching this model, we will reach a model known as the *Turing machine*, different versions of which will be seen to define the *computable* languages and the languages considered to be efficiently computable.

The interplay between computational models and the language classes that they define is an overarching theme of this book. In particular, for each of the various models, we endeavor to understand the range of its language class, which yields insight into the nature and capabilities of the model. At the same time, we develop tools so that, when confronted with a language of interest, we may attempt to classify it within our taxonomy by trying to understand which classes it does and does not fall into. In essence, performing such classification makes precise what form of computing machinery is needed, or not needed, to cope with the language and its accompanying decision problem.

Agreements

Here, we present some definitions that will be basic for our study; we also set down some of the conventions to be used during our presentation.

Alphabets, strings, and languages

An **alphabet** is a nonempty finite set, typically denoted by capital Greek letters such as Σ and Γ. Here are three examples of alphabets:

- $\Sigma_1 = \{0, 1\}$,
- $\Sigma_2 = \{0, 1, 2, \ldots, 9\}$,
- $\Sigma_3 = \{a, b, c\}$.

We refer to the elements of an alphabet as **symbols**. We tend to use the term *alphabet* to refer to a set having the specified properties when we form *strings* over the set.

A **string** over an alphabet Σ is a finite-length sequence of symbols from Σ; the **length** of a string x is its length as a sequence, and is denoted by $|x|$. As examples:

- *abbaba* is a string of length 6 over the alphabet $\{a, b\}$,
- 31415926 is a string of length 8 over the alphabet $\{0, 1, 2, \ldots, 9\}$.

By convention, there is a unique string of length 0 (over any alphabet), which is called the **empty string** and denoted ϵ. It is always assumed that ϵ does not occur as a symbol in an alphabet; that is, for each alphabet Σ, we assume that $\epsilon \notin \Sigma$. Note that we write the symbols of a string contiguously, without any separating marker. When x is a string of length m, we often use $x_1, \ldots, x_m$ to denote its constituent symbols, so that $x = x_1 \ldots x_m$.

When $x = x_1 \ldots x_m$ is a string of length m and $y = y_1 \ldots y_n$ is a string of length n, the **concatenation** of x and y is the string $x_1 \ldots x_m y_1 \ldots y_n$ of length $m + n$, and is denoted by xy or $x \cdot y$. Observe that for any string x, it holds that $\epsilon x = x\epsilon = x$. When x is a string and $k \geq 0$, we use the exponentiation notation x^k to denote the concatenation of x with itself k times:

$$x^k = \underbrace{x \cdot x \cdot \cdots \cdot x}_{k}.$$

By a usual and useful convention, for any string x, we consider x^0 to be the empty string ϵ. By default, in working with strings, exponentiation is evaluated prior to other concatenation. So for example, over the alphabet $\Sigma = \{a, b\}$, we have $bab^2a^3a = babbaaaa$.

A string x is a **prefix** of another string y if there exists a string v such that $y = xv$. A string x is a **substring** of another string y if there exist strings u, v such that $y = uxv$. For example, consider the string *abcd*; its prefixes are ϵ, a, ab, abc, and $abcd$, and its length 2 substrings are ab, bc, and cd. Observe that for any string y, it holds that each of the strings ϵ and y is both a prefix and a substring of y, and indeed it holds that each prefix of y is a substring of y.

When x is a string over alphabet Σ and $a \in \Sigma$, we use the notation $\#_a(x)$ to denote the number of occurrences of the symbol a in the string x. Over the alphabet $\Sigma = \{0, 1\}$, for example, we have $\#_0(01101) = 2$, $\#_1(01101) = 3$, $\#_0(10^51^6) = 5$, and $\#_1(10^51^6) = 7$.

When Σ is an alphabet, we use Σ^* to denote the set of all strings over Σ. As examples:

- For $\Sigma_4 = \{a, b\}$, we have $\Sigma_4^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$.
- For $\Sigma_5 = \{0\}$, we have $\Sigma_5^* = \{\epsilon, 0, 00, 000, \ldots\}$.

In both cases, we have explicitly presented some initial elements of Σ^* according to a length-increasing ordering. We note that, when Σ is an alphabet, the set Σ^* is always countably infinite.

A **language** over an alphabet Σ is a set of strings over Σ; equivalently, a **language** over Σ is a subset of Σ^*. When B is a language over alphabet Σ, its **complement**, denoted by $\overline{B}$, is defined as $\Sigma^* \setminus B$, that is, as the complement of B with respect to Σ^*; whenever we refer to the complement of a language, the alphabet Σ should be clear from the context. A language B over alphabet Σ is **trivial** if $B = \emptyset$ or $B = \Sigma^*$, and is **nontrivial** otherwise. Observe that a language is trivial if and only if its complement is.

Conventions

Here, we present mathematical notions and notation to be used throughout the book.

We use the notation $\mathbb{N}$ to denote the set of **natural numbers** $\{0, 1, 2, \ldots\}$. A natural number is **positive** if it is not equal to 0; we use the notation $\mathbb{N}^+$ to denote the set of positive natural numbers $\{1, 2, 3, \ldots\}$. By default, we assume all numbers under discussion to be natural numbers, unless mentioned otherwise. By a **unary representation** or a **unary encoding** of a natural number n, we refer to a string c^n containing n occurrences of a symbol c; typically, c is taken to be the symbol 1. For our purposes, a **multiple** of a natural number $d \in \mathbb{N}$ is a natural number that can be expressed in the form $d \cdot k$ where $k \in \mathbb{N}$; here, with $d \cdot k$ we denote the product of d and k. As an example, the five initial multiples of 4 are 0, 4, 8, 16, and 20. When d and n are natural numbers, we say that d is a **divisor** of n if n is a multiple of d, and that d is a **proper divisor** of n if, in addition, it holds that $1 < d < n$. A **prime number** is defined as a natural number that is greater than or equal to 2 and that has no proper divisor.

When B and C are sets, B is a **subset** of C if each element of B is an element of C; B is a **proper subset** of C if, in addition, B is not equal to C. We write $B \subseteq C$ to indicate that B is a subset of C, and $B \subsetneq C$ to indicate that B is a proper subset of C.

The **power set** of a set C is denoted by $\wp(C)$, and is defined as the set containing as elements all subsets of C; that is, $\wp(C) = \{B \mid B \subseteq C\}$. For example, we have

$$\wp(\{1,2,3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

Observe that, for any set C, the empty set $\emptyset$ and C itself are both elements of $\wp(C)$. When C is a set, we use $\wp_{\text{fin}}(C)$ to denote $\{B \mid B \subseteq C \text{ and } B \text{ is a finite set}\}$, that is, the subset of $\wp(C)$ whose elements are the finite sets in $\wp(C)$.

The **product** of two sets B and C, denoted $B \times C$, is the set of pairs (b, c) where the first coordinate b is an element of B, and the second coordinate c is an element of C. That is,

$$B \times C = \{(b, c) \mid b \in B, c \in C\}.$$

For example,

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

More generally, the **product** of a finite sequence of sets $B_1, \ldots, B_k$, denoted $B_1 \times \cdots \times B_k$, is the set of tuples

$$B_1 \times \cdots \times B_k = \{(b_1, \ldots, b_k) \mid b_1 \in B_1, \ldots, b_k \in B_k\}.$$

Note that tuples are considered to be ordered; so, for example, $(1,2)$ and $(2,1)$ are considered to be distinct tuples.

Let B be a set and let k be a natural number. We use B^k to denote the k-fold product

$$\underbrace{B \times \cdots \times B}_{k}.$$

For any set B, we consider B^0 to be the set containing a single tuple, called the **empty tuple.** A **k-ary relation** on B is defined as a subset of B^k. A 2-ary relation is also called a **binary relation**. When R is a binary relation on B, we will sometimes use the infix notation aRb to indicate that $(a, b) \in R$. Examples of binary relations on a set B include the empty set $\emptyset$; the equality relation on B, which is the set $\{(b, b) \mid b \in B\}$; and, the set $B \times B$.

Let $f : A \to B$ be a function. We use $f[c \mapsto d]$ to denote the function that maps c to d, and otherwise behaves as f does, mapping each element $a \in A \setminus \{c\}$ to $f(a)$. On occasion, we extend this notation, by using $f[c_1 \mapsto d_1, \ldots, c_k \mapsto d_k]$ to denote the function that maps each c_i to d_i, and maps each element in $a \in A \setminus \{c_1, \ldots, c_k\}$ to $f(a)$. Whenever this extended notation is used, the values $c_1, \ldots, c_k$ will be pairwise distinct.

1 Automata Theory

One for sorrow,
Two for mirth,
Three for a wedding,
Four for birth . . .
— Traditional nursery rhyme

Our story commences with the study of *finite-state automata*, computational models that are quite restricted in that each *automaton* can only use a bounded amount of memory and processes an input string by reading it once from left to right. Although relatively simple, they will allow us to encounter and explore, in a gentle fashion, a number of the themes that will recur in our study of computation—for instance, they will provide our first exposure to nondeterminism, a concept at the heart of the *P versus NP* question in complexity theory. They also enjoy applications, for example, in text searching and parsing. Moreover, they give rise to a theory that is elegant and appealing in its own right.

1.1 Deterministic finite automata

We begin by presenting our first computational model: the *deterministic finite automaton (DFA)*. A DFA contains a finite set of *states*, which represent its only memory; one state is designated the *start state*. Given an input string, a DFA begins in its start state and reads in one symbol of the string at a time. Each time a symbol is read, the automaton discretely changes state based on its current state and the read symbol; the way in which the state is changed is specified by a *transition function*. After having read all symbols of a string, a DFA accepts or rejects the string based on whether or not its final state is an *accept state*. We proceed to the formal definition of this model.

Definition 1.1.1. A **deterministic finite automaton (DFA)** is a 5-tuple $M = (Q, \Sigma, s, T, \delta)$ where

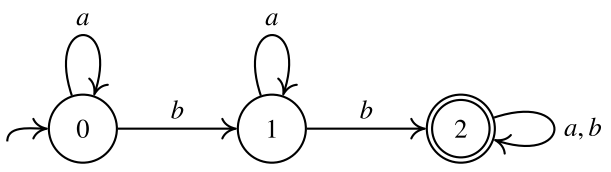
- Q is a finite set called the **state set**, whose elements are called **states**,
- Σ is an alphabet called the **input alphabet**,
- $s \in Q$ is a state called the **start state** or **initial state**,
- $T \subseteq Q$ is a set of states, where each member is called an **accept state**, and
- $\delta \colon Q \times \Sigma \to Q$ is a function called the **transition function**. ◇

To get a feel for this model, we consider some examples.

Example 1.1.2. As a first example of a DFA, take the set of states to be $Q = \{0, 1, 2\}$; the input alphabet to be $\Sigma = \{a, b\}$; the initial state s to be 0; and the set of accept states T to be $\{2\}$. We give the transition function δ by the following table:

δ	a	b
0	0	1
1	1	2
2	2	2

When specifying a DFA, we must specify all five of its parts! We have just done this by giving each of the parts individually. There is another convenient and often intuitive way to specify a DFA, namely, drawing a diagram. The following is a diagram for the example DFA just given, drawn under the conventions we will use:



In general, we form the diagram for a DFA as follows. Each state is placed in a circle; the initial state is indicated by an unlabeled arrow that points to it; and each accept state has a double circle placed around it. Each *transition* $\delta(p, c) = q$ is indicated by drawing an arrow from the state p to the state q, with label c. Multiple transitions having the same source and target states are indicated using the same arrow, but with multiple labels; for example, in the diagram above, the transitions $\delta(2, a) = 2$ and $\delta(2, b) = 2$ are indicated by a single arrow from the state 2 to itself having the two labels a and b.

Let us explain how this DFA processes strings. As an opening example, consider the string bab. The DFA begins in its start state 0. It reads the initial symbol b, and makes a transition to the state $\delta(0, b) = 1$, that is, the state that the transition function yields when fed the current state with the seen symbol. Once in state 1, the DFA then reads the second symbol a, and makes a transition to the state $\delta(1, a) = 1$, so it effectively stays in the same state. It then reads the final symbol b, and makes a transition from state 1 to the state $\delta(1, b) = 2$. At this point, the DFA has fully processed the string, and has ended up in the state 2, which is an accept state; thus, the string *bab* is said to be *accepted*. As another example, consider the string *aba*. To process this string, the DFA begins in state 0; it reads the initial symbol a and remains in state 0; it then reads the second symbol b and transitions to state 1; and it then reads the final symbol a and terminates in state 1. As state 1 is not an accept state, the string *aba* is said to be *rejected*. Verify further for yourself that the strings *ab* and *ba* are rejected, and that the string *abba* is accepted.

There is a simple description of the strings that are accepted by this DFA; to arrive at it, let us contemplate the transition function. When this DFA reads the symbol a, it does not change state. When this DFA reads the symbol b, from state 0 or 1, it increments its state by one, proceeding to state 1 or 2, respectively; from state 2, it remains in state 2. In effect, the state of the DFA counts the number of b symbols that it has seen so far, up to 2; once it reaches the state 2, it remains there. As this DFA only accepts strings that cause it to terminate in state 2, it accepts precisely each string that contains two or more occurrences of the symbol b. ◇

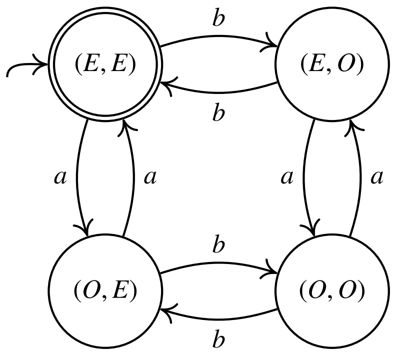
Example 1.1.3. We present a second example of a DFA, where each state is a pair. This DFA has state set

$$Q = \{(E, E), (E, O), (O, E), (O, O)\},$$

input alphabet $\Sigma = \{a, b\}$, initial state $s = (E, E)$, and set of accept states $T = \{(E, E)\}$, so the initial state is the unique accept state. The following table gives the transition function:

δ	a	b
(E, E)	(O, E)	(E, O)
(E, O)	(O, O)	(E, E)
(O, E)	(E, E)	(O, O)
(O, O)	(E, O)	(O, E)

A diagram for this DFA is as follows:



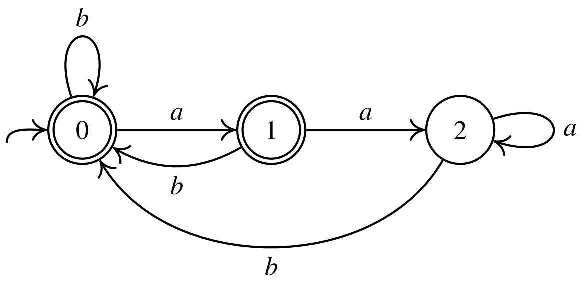
What does this DFA do? Each of its states consists of two components; each of these components can be either E or O. Whenever the symbol a is read, the first component toggles between E and O; similarly, whenever the symbol b is read, the second component toggles between E and O. In effect, the first component keeps track of whether or not the number of a's seen is **e**ven or **o**dd, and the second component keeps track of whether or not the number of b's seen is **e**ven or **o**dd. (The even natural numbers are $0, 2, 4, \ldots$ and the odd natural numbers are $1, 3, 5, \ldots$.) When the DFA has not yet read any symbols, both the number of a's seen and the number of b's seen are equal to 0, an even number; this observation accords with the initial state being (E, E). Since the only accept state is (E, E), a string is accepted by this DFA if and only if its number of occurrences of a and its number of occurrences of b are both even. ◇

Remark 1.1.4. The particular names given to states in a DFA are, in a sense, immaterial to the DFA's functioning. Suppose that a DFA is modified by renaming its states, and adjusting its other parts correspondingly; in terms of the DFA's diagram, this amounts to just changing the name of each state inside each state's circle. Then, on any input string, the modified DFA makes transitions corresponding to those of the original DFA, and accepts a string if and only if the original DFA does. ◇

Example 1.1.5. As another example, consider the DFA having state set $Q = \{0, 1, 2\}$, input alphabet $\Sigma = \{a, b\}$, initial state $s = 0$, accept states $T = \{0, 1\}$, and the following transition function:

δ	a	b
0	1	0
1	2	0
2	2	0

The following is a diagram for this DFA:



Let us consider how this DFA treats a few example strings. On the string aa, the DFA terminates in state 2, and rejects. On the string aba, the DFA terminates in state 1, and accepts. And on the string $aaaba$, the DFA also terminates in state 1, and accepts.

Which strings are accepted by this DFA? When the symbol a is read in state 0 or 1, this DFA increments its state by 1; when the symbol a is read in state 2, this DFA remains in state 2. When the symbol b is read, however, the DFA always resets its state to 0. Following these observations, it can be seen that the DFA will be in state 2 precisely when the last two symbols read are a, that is, if the string read thus far ends with aa. This DFA, however, accepts if it terminates in state 0 or 1; therefore, it accepts a string if and only if the string does *not* end with aa. ◇

We now introduce the notion of a *configuration* of a DFA, which will be very useful to reason about DFA behavior. In fact, as our study proceeds, we will define a notion of *configuration* for each computational model to be considered. In general, a configuration captures everything about a computation, at a given moment in time, that is relevant to know how the computation will proceed. In the case of a DFA, this amounts to the state that the DFA is in, and the portion of the string that has *not* yet been read.

We also introduce the notion of the *initial configuration* of a DFA on a string, and the *successor configuration* of a configuration of a DFA, which formalizes how the DFA processes a single symbol.

Definition 1.1.6. Let $M = (Q, \Sigma, s, T, \delta)$ be a DFA.

- A **configuration** of M is a pair $[q, y]$ consisting of a state $q \in Q$ and a string $y \in \Sigma^*$.
- The **initial configuration** of M on a string $y \in \Sigma^*$ is the configuration $[s, y]$.
- The **successor configuration** of a configuration $[q, y]$ of M is defined when $|y| \geq 1$ (that is, when y is not the empty string ϵ); in this case, denoting y by ax with $a \in \Sigma$ and $x \in \Sigma^*$, the configuration $[\delta(q, a), x]$ is the successor configuration of $[q, y]$. ◇

So relative to a DFA, when the successor configuration of a configuration $[q, y]$ is defined, it is obtained by removing the string y's leftmost symbol a, and replacing the state q with the state $\delta(q, a)$.

Example 1.1.7. Consider the DFA M from Example 1.1.2. The initial configuration of M on the string bab is $[0, bab]$. The successor configuration of the configuration $[0, bab]$ is $[1, ab]$; the successor of $[1, ab]$ is $[1, b]$; the successor of $[1, b]$ is $[2, \epsilon]$; and the configuration $[2, \epsilon]$ has no successor configuration, as its string is the empty string. ◇

We next introduce notation to discuss configurations of a DFA M; in particular, we introduce binary relations on the set of configurations. However, before proceeding to this, a remark is in order. A configuration of a DFA has at most one successor configuration; we accordingly speak of *the* successor configuration of a configuration of a DFA. In the following definition, however, we speak of *a* successor configuration of a configuration; this is because we will want to reuse this definition, and employ it to discuss other computational models (in particular, *nondeterministic* models) where a configuration may have more than one successor configuration.

Definition 1.1.8. Let α and β be configurations of M.

- We write $\alpha \vdash_M \beta$ if β is a successor configuration of α.
- For each $n \geq 0$, we write $\alpha \vdash_M^n \beta$ if β can be obtained by starting from α and iteratively taking a successor configuration n times. That is, we write $\alpha \vdash_M^n \beta$ if there exist configurations $\gamma_0, \gamma_1, \ldots, \gamma_n$ such that $\gamma_0 = \alpha$, $\gamma_n = \beta$, and $\gamma_0 \vdash_M \gamma_1 \vdash_M \cdots \vdash_M \gamma_n$.
- We write $\alpha \vdash_M^* \beta$ if there exists $n \geq 0$ such that $\alpha \vdash_M^n \beta$. (Note that we can view the relation $\vdash_M^*$ as the union $\bigcup_{n \geq 0} \vdash_M^n$ of the relations $\vdash_M^n$.)

In using this notation, we sometimes omit the M subscript when the context allows. ◇

Remark 1.1.9. In Definition 1.1.8, the relations $\vdash_M^n$ may be equivalently defined by induction, as follows:

- It holds that $\alpha \vdash_M^0 \beta$ if and only if $\alpha = \beta$.
- For each $n > 0$, it holds that $\alpha \vdash_M^n \beta$ if and only if there exists a configuration γ such that both $\alpha \vdash_M \gamma$ and $\gamma \vdash_M^{n-1} \beta$ hold. ◇

Example 1.1.10. Consider again the DFA M from Example 1.1.2. Let us continue the discussion of Example 1.1.7; from the observations made there, we may write the following:

$$[0, bab] \vdash_M [1, ab] \vdash_M [1, b] \vdash_M [2, \epsilon].$$

Having seen this, we may give the following examples of the notation presented in Definition 1.1.8:

$$\begin{aligned}
[1, ab] &\vdash_M^0 [1, ab], & [0, bab] &\vdash_M^3 [2, \epsilon], \\
[1, ab] &\vdash_M^1 [1, b], & [1, b] &\vdash_M^* [1, b], \\
[1, b] &\vdash_M^1 [2, \epsilon], & [1, ab] &\vdash_M^* [2, \epsilon], \\
[0, bab] &\vdash_M^2 [1, b], & [0, bab] &\vdash_M^* [1, b], \\
[1, ab] &\vdash_M^2 [2, \epsilon], & [0, bab] &\vdash_M^* [2, \epsilon].
\end{aligned}$$
◇

We will often refer to a particular realization of a computational model—for example, a particular DFA—as a **machine**. In general, we refer to the process by which a machine operates on an input string as a **computation**. We also use the term **computation** to refer in particular to a sequence containing all configurations that a machine passes through when invoked on an input string; in the case of a DFA, such a sequence begins with an initial configuration, and ends with a configuration having no successor.

Example 1.1.11. Let M again be the DFA from Example 1.1.2. The following are examples of computations of this DFA M:

$$[0, bab] \vdash_M [1, ab] \vdash_M [1, b] \vdash_M [2, \epsilon],$$
$$[0, aba] \vdash_M [0, ba] \vdash_M [1, a] \vdash_M [1, \epsilon].$$
◇

Let $M = (Q, \Sigma, s, T, \delta)$ be a DFA, let y be a string of length n, and let $q \in Q$ be a state. Starting from the configuration $[q, y]$, we may iteratively take the successor configuration n times; in doing so, each time we take the successor configuration, it is unique, and its string is the string of its predecessor configuration with the first symbol removed. We thus have the following observation, which we will use tacitly throughout our discussion: for each value k with $0 \leq k \leq n$, there is a unique configuration β such that $[s, y] \vdash_M^k \beta$, and the string component of β is equal to y with its first k symbols removed.

We may now define formally what it means for a string to be *accepted* or *rejected* by a DFA. This status is determined by the state that the DFA arrives at after processing the string, when it begins from the respective initial configuration.

Definition 1.1.12. Let $M = (Q, \Sigma, s, T, \delta)$ be a DFA. Let $y \in \Sigma^*$ be a string of length n, and let $q \in Q$ be the unique state such that $[s, y] \vdash_M^n [q, \epsilon]$.

- If $q \in T$, we say that M **accepts** y.
- If $q \notin T$, we say that M **rejects** y.

We define the **language of** M, denoted by $L(M)$, to be $\{y \in \Sigma^* \mid M \text{ accepts } y\}$. ◇

Although it may seem that we are merely formalizing notions that are intuitively clear, there are at least a couple of reasons why we want to make fully precise and formal the components and behavior of a DFA. First, a true formalization will allow us to rigorously prove theorems and results about DFA, for example, impossibility results demonstrating the limitations of DFA. We would be hard-pressed to prove limitations on a computational model that was not well-defined! Second, the process of formalization that we have carried out for DFA offers us gainful practice and preview for the study that follows, in which we will formalize increasingly complex computational models—not all of which are, by any means, as simple or transparent as the DFA.

As we have just seen, each DFA M gives rise to a language $L(M)$, which we call the **language of** M or the **language decided by** M. We will want to discuss in an aggregate fashion all of the languages thusly arising, and hence give the following name to a language decided by a DFA.

Definition 1.1.13. A language B is **regular** if there exists a DFA M where $B = L(M)$. ◇

That is, a language is regular if there is *some* DFA that decides it. We have here identified a class of languages: each language is either regular, or it is not. Having been provided this definition, perhaps the most basic question that one could proceed to ask is whether or not there is a language that is *not* regular. (If there is no such language, our definition would be somehow trivial: in this case, the identified class of languages would simply be the class of all languages.) At the risk of quashing the suspense, it can be reported here that there are indeed languages that are not regular. Perhaps the most classic example of a language that is not regular is

$$\{a^n b^n \mid n \geq 0\} = \{\epsilon, ab, aabb, aaabbb, \dots\}.$$

In words, a string is in this language if it begins with some number of a's, followed by the same number of b's. Later in this chapter, we will acquire techniques for proving that this language and others are not regular.[1]

1.2 Closure properties

We just defined the class of regular languages in terms of the DFA computational model. As we encounter further computational models, we will correspondingly define further classes of languages. One basic type of question that we will ask, for each of these various classes of languages, is whether or not they possess certain *closure properties*. For example, in a moment we will ask whether or not the regular languages are *closed under complementation*, that is, whether or not the complement of an arbitrary regular language is always itself regular. Given a class of languages, one can indeed inquire about closure under any operation defined on languages: the class is closed under such an operation if, whenever the operation is applied to a language or languages from the class, the resulting language is also in the class. Understanding the closure properties of a class of languages gives us insight into the internal structure of the class, and can be helpful in identifying whether a particular language is inside or outside of the class.

In this section, to show closure properties of the regular languages, we demonstrate that operations on regular languages can be effected by performing respective operations on DFA. This pattern exemplifies our general approach to showing closure properties of language classes: the typical language class that we study is defined from a group of machines, and so establishing closure properties on the class is naturally performed by defining operations on the machines.

1.2.1 Complementation

We begin by considering the operation of complementation. What would it mean for the regular languages to be closed under complementation? Let us recall that a language is regular if there exists a DFA that decides it. Hence, closure under complementation would mean that for an arbitrary DFA M, it is possible to design a second DFA M' that decides the complement of $L(M)$. By definition of the complement, the DFA M' should reject each string that is accepted by M, and accept each string that is rejected by M: its final judgment should always be the polar opposite of that of M. It is indeed always possible to design such a DFA M', by starting with M, and swapping the acceptance status of each of the states: a state is accepting in M' if and only if it was not accepting in M.

Theorem 1.2.1. *If B is a regular language over the alphabet Σ, then its complement $\overline{B}$ is also a regular language.*

1. Intuitively speaking, in order to check if a string is in this language by scanning the string from left to right, it is necessary to first count the number n of a's that occurs, and then ensure that the number of b's that follow is equal to n; a DFA, however, cannot count up to an arbitrary natural number, and hence cannot decide this language.

Proof. Since B is a regular language, there exists a DFA $M = (Q, \Sigma, s, T, \delta)$ such that $L(M) = B$. Define $M' = (Q, \Sigma, s, T', \delta)$ be the DFA that is identical to M except that its set T' of accept states is $Q \setminus T$.

We claim that $L(M') = \overline{B}$. The DFA M and M' have the same set of states and the same transition function; by a review of Definition 1.1.6, one sees that they have the same configurations and also the same notion of successor configuration, that is, the relations $\vdash_M$ and $\vdash_{M'}$ are equal. So, for any string $x \in \Sigma^*$ of length n, if we let $q \in Q$ be the unique state such that $[s, x] \vdash_M^n [q, \epsilon]$, then $[s, x] \vdash_{M'}^n [q, \epsilon]$ also holds. We have that $q \in T$ if and only if $q \notin T'$, so x is accepted by M if and only if x is rejected by M', yielding the claim. □

1.2.2 Intersection and union

We next consider closure under intersection; the regular languages satisfy this closure property, in the following formalization.

Theorem 1.2.2. *If B and C are both regular languages over the same alphabet Σ, then their intersection $B \cap C$ is also a regular language.*

Let $M_B = (Q_B, \Sigma, s_B, T_B, \delta_B)$ be a DFA with $L(M_B) = B$, and let $M_C = (Q_C, \Sigma, s_C, T_C, \delta_C)$ be a DFA with $L(M_C) = C$. To establish the theorem, our mission is to construct a DFA that decides $B \cap C$. How are we to do this? In particular, what should the state set of our new DFA be? A natural idea is the following: as the new DFA processes a string, its state keeps track of both the state that the first DFA M_B would be in, as well as the state that the second DFA M_C would be in. This can be accomplished naturally by taking the state set of the new DFA to be the product $Q_B \times Q_C$ of the state sets of the original two DFA.

In a construction typically referred to as the **product construction**, we use the DFA M_B and M_C to define a DFA $M = (Q, \Sigma, s, T, \delta)$, as follows:

$$Q = Q_B \times Q_C,$$

$$s = (s_B, s_C),$$

$$T = T_B \times T_C,$$

$$\delta((q_B, q_C), a) = (\delta_B(q_B, a), \delta_C(q_C, a)).$$

Why do the definitions of the other parts make sense? The start state s should indicate where each of the original DFA start; hence, we take the pair consisting of the start states of those DFA. The new DFA M should accept a string precisely when both of the original DFA accept the string. Hence, its set of accept states should contain all state pairs such that the first state is accepting in M_B, and the second state is accepting in M_C; this can be expressed as the product $T_B \times T_C$. Finally, when a symbol is read and the new DFA M is in the state (q_B, q_C), the first component q_B should be updated according to the transition function δ_B, and analogously the second component q_C should be updated according to the transition function δ_C. Figure 1.2.1 provides an example of this construction.

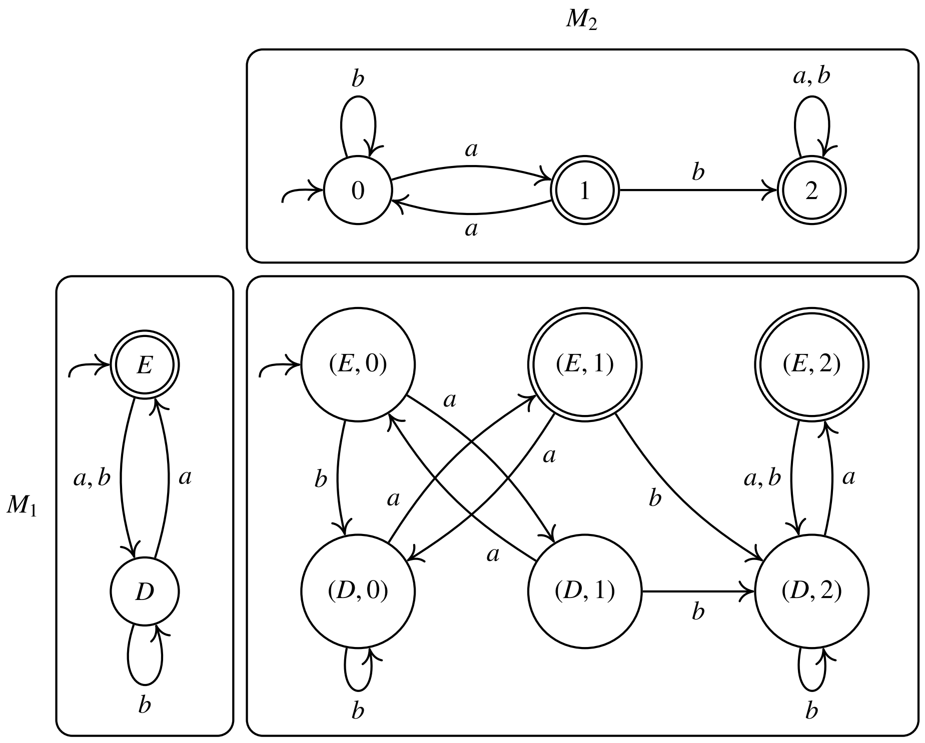


Figure 1.2.1. Example of the product construction on DFA. Here, the construction is applied to the shown DFA $M_1 = (Q_1, \Sigma, s_1, T_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, s_2, T_2, \delta_2)$; each is over the alphabet $\Sigma = \{a, b\}$. The state sets of these two DFA are $Q_1 = \{E, D\}$ and $Q_2 = \{0, 1, 2\}$; the state set of the resulting DFA is $Q_1 \times Q_2$. The start state of the resulting DFA is the pair $(E, 0)$ obtained by pairing the start states of the original two DFA. The set of accept states of the resulting DFA is the product $T_1 \times T_2$, which is equal to $\{E\} \times \{1, 2\} = \{(E, 1), (E, 2)\}$. The transition function δ of the resulting DFA, when applied to a pair and a symbol, is defined to yield the applications of the transition functions δ_1 and δ_2 to the respective pair entries, along with the symbol; as one example transition according to δ, we have $\delta((E, 1), b) = (\delta_1(E, b), \delta_2(1, b)) = (D, 2)$.

In order to prove, as desired, that the language of M is the intersection $B \cap C$, we first establish a lemma showing that M behaves as claimed: from a state (q_B, q_C), after a string y is processed, the resulting state is the pair consisting of

- the state that M_B would end up in after processing y from q_B, and
- the state that M_C would end up in after processing y from q_C.

Lemma 1.2.3. *Let $y \in \Sigma^*$ be a string of length n; let $q_B \in Q_B$ and $q_C \in Q_C$ be arbitrary states; and let $r_B \in Q_B$ and $r_C \in Q_C$ be the unique states where $[q_B, y] \vdash^n_{M_B} [r_B, \epsilon]$ and $[q_C, y] \vdash^n_{M_C} [r_C, \epsilon]$. Then, it holds that $[(q_B, q_C), y] \vdash^n_M [(r_B, r_C), \epsilon]$.*

Proof. We prove this by induction on n.

When $n = 0$, we have $y = \epsilon$, $q_B = r_B$, and $q_C = r_C$, from which the claim can be seen.

When $n > 0$, write $y = ax$ where $a \in \Sigma$ and $x \in \Sigma^*$; Figure 1.2.2 shows a diagram indicating the setup and the result for this case. Define

$$q'_B = \delta_B(q_B, a) \quad \text{and} \quad q'_C = \delta_C(q_C, a).$$

We have

$$[q_B, ax] \vdash_{M_B} [q'_B, x] \vdash^{n-1}_{M_B} [r_B, \epsilon] \quad \text{and} \quad [q_C, ax] \vdash_{M_C} [q'_C, x] \vdash^{n-1}_{M_C} [r_C, \epsilon].$$

We also have, from the definition of δ, that

$$[(q_B, q_C), ax] \vdash_M [(q'_B, q'_C), x].$$

By appeal to induction, we obtain from $[q'_B, x] \vdash^{n-1}_{M_B} [r_B, \epsilon]$ and $[q'_C, x] \vdash^{n-1}_{M_C} [r_C, \epsilon]$ that

$$[(q'_B, q'_C), x] \vdash^{n-1}_M [(r_B, r_C), \epsilon].$$

Combining the previous two results, we obtain $[(q_B, q_C), ax] \vdash^n_M [(r_B, r_C), \epsilon]$. □

Proof of Theorem 1.2.2. Let $x \in \Sigma^*$ be a string of length n. By Lemma 1.2.3, when we define r_B and r_C to be the states such that $[s_B, x] \vdash^n_{M_B} [r_B, \epsilon]$ and $[s_C, x] \vdash^n_{M_C} [r_C, \epsilon]$, we obtain $[(s_B, s_C), x] \vdash^n_M [(r_B, r_C), \epsilon]$. We then have

$$\begin{aligned}
x \in B \cap C &\Leftrightarrow x \in L(M_B) \cap L(M_C) && \text{(by the choices of } M_B \text{ and } M_C\text{)}\\
&\Leftrightarrow r_B \in T_B \text{ and } r_C \in T_C && \text{(by the definition of acceptance for } M_B \text{ and } M_C\text{)}\\
&\Leftrightarrow (r_B, r_C) \in T_B \times T_C && \text{(by the definition of the set product } T_B \times T_C\text{)}\\
&\Leftrightarrow (r_B, r_C) \in T && \text{(by the definition of } T\text{)}\\
&\Leftrightarrow x \in L(M) && \text{(by the definition of acceptance for } M\text{).}
\end{aligned}$$
□

We next consider closure under union; once again, we have that the class of regular languages enjoys this closure property.

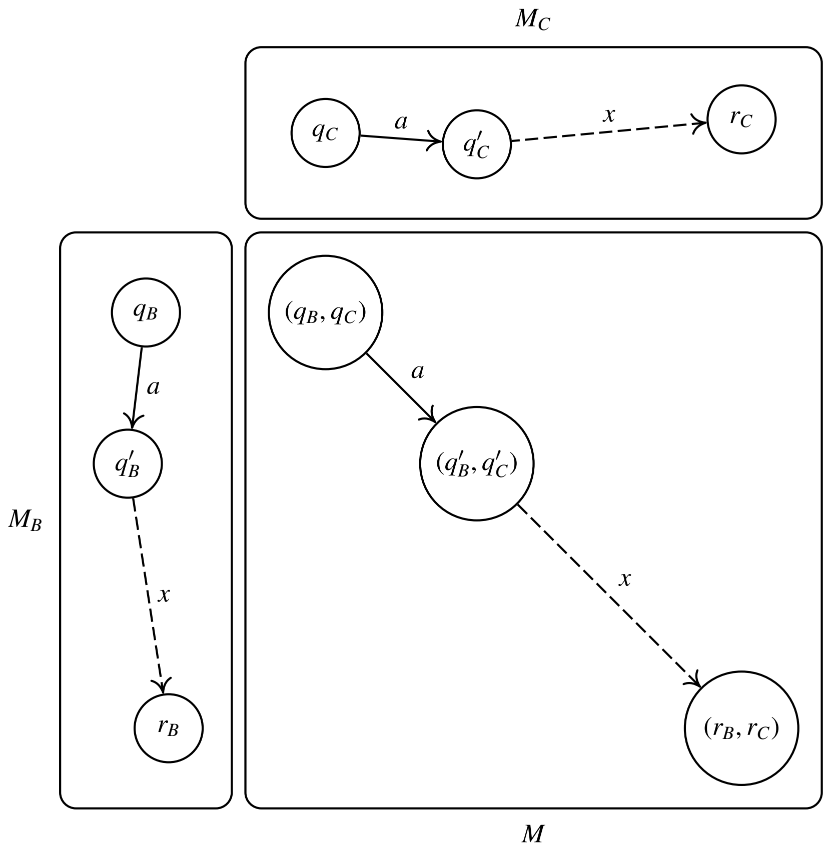


Figure 1.2.2. The setup and desired result of the inductive step in the proof of Lemma 1.2.3. The string y is viewed as the concatenation of a and x, where $a \in \Sigma$ is a single symbol and $x \in \Sigma^*$ is a string. In the DFA M_B, from the state q_B, reading the symbol a leads to the state q'_B, and then reading the string x leads to the state r_B. Similarly, in the DFA M_C, from the state q_C, reading the symbol a leads to the state q'_C, and then reading the string x leads to the state r_C. It follows that, in the DFA M, from the state (q_B, q_C), reading the symbol a leads to the state (q'_B, q'_C), and by induction, subsequently reading the string x leads to the state (r_B, r_C).

Theorem 1.2.4. *If B and C are both regular languages over the same alphabet Σ, then their union $B \cup C$ is also a regular language.*

At this point, there are a couple of ways that we may prove this theorem.

One way to proceed is to directly present a DFA, as was done for the previous theorem. Namely, we can directly describe, given two DFA M_B and M_C, a DFA M' whose language $L(M')$ is equal to $L(M_B) \cup L(M_C)$. Indeed, such a DFA M' may be defined as identical to the DFA M above, but with the change that its set of accept states is

$$T' = \{(q_B, q_C) \in Q \mid q_B \in T_B \text{ or } q_C \in T_C\}.$$

Note that the accept states of the DFA M, namely $T = T_B \times T_C$, may be equivalently expressed as

$$T = \{(q_B, q_C) \in Q \mid q_B \in T_B \text{ and } q_C \in T_C\}.$$

One can see that, to define T', the *and* in this expression of T has been changed to *or*, reflecting the difference in definition between the intersection and the union. Lemma 1.2.3 holds with M' in place of M, as its statement and its proof do not refer to the set of accept states of M. By adjusting the argumentation in the proof of Theorem 1.2.2, it can be proved that $L(M') = L(M_B) \cup L(M_C)$. (We leave a verification of this to the reader.)

We may alternatively obtain that the regular languages are closed under union by invoking the following versions of De Morgan's laws.

Proposition 1.2.5 (De Morgan's laws, for languages). *For any languages B and C over the same alphabet, the following hold:*

(1) $B \cup C = \overline{(\overline{B} \cap \overline{C})}$,

(2) $B \cap C = \overline{(\overline{B} \cup \overline{C})}$.

We have already established that the regular languages are closed under intersection and complement. So, when B and C are regular languages, the languages $\overline{B}$ and $\overline{C}$ are also regular, implying that $\overline{B} \cap \overline{C}$ is regular, from which we obtain that $\overline{(\overline{B} \cap \overline{C})}$ is regular. By De Morgan's law (1), this immediately implies that $B \cup C$ is regular.

Note that De Morgan's law (1) implies that, in general, *any* class of languages closed under intersection and complement is closed under union. Similarly, the dual De Morgan's law (2) implies that any class of languages closed under union and complement is closed under intersection.

1.3 Nondeterministic finite automata

Nondeterminism in computation is a theoretical construct; it is not intended to faithfully model real computers or any aspect thereof, but rather is an instrument for analysis. In deterministic computation models, such as the DFA, how a computation evolved was uniquely determined at each step: as long as the computation proceeded, each configuration had a unique successor configuration. In nondeterministic computation models, a configuration may have multiple successor configurations, and acceptance is always defined via the notion of *possibility*: a string is accepted by a nondeterministic machine if there *exists* a computation which results in acceptance. This section introduces nondeterministic finite automata (NFA), which are nondeterministic counterparts of the deterministic finite automaton. Whereas a deterministic finite automaton has a transition function that provides a *unique* state, given a state and a symbol, a nondeterministic finite automaton has a transition function that provides a *set* of states, given a state and a symbol.

As a theoretical construct, nondeterminism has been immensely and supremely fruitful in supplying insights into the nature of computation; in particular, it has shed light and perspective on the reach and limitations of deterministic computation. In our study of complexity theory, nondeterministic computation will be used crucially to classify languages of interest. By the end of the current section, we will have compared NFA to DFA formally and will have shown that these two models have the same expressiveness (in a sense made precise). An offshoot of this result is that providing an NFA for a language is an avenue for establishing the language's regularity. Working with NFA has the advantages that they may be more succinct than DFA, and also that they may be easier to comprehend and maintain.

In this section, we present and study two brands of nondeterministic automata: the *nondeterministic finite automaton (NFA)*, and an extension thereof referred to as the *ϵ-NFA*.

1.3.1 NFA

We begin with the definition of NFA.

Definition 1.3.1. A **nondeterministic finite automaton (NFA)** is a 5-tuple $M = (Q, \Sigma, S, T, \Delta)$ where

- Q is a non-empty finite set called the **state set**, whose members are called **states**,
- Σ is an alphabet called the **input alphabet**,
- $S \subseteq Q$ is a set of states, where each member is called a **start state** or an **initial state**,
- $T \subseteq Q$ is a set of states, where each member is called an **accept state**, and
- $\Delta \colon Q \times \Sigma \to \wp(Q)$ is a function called the **transition function**. ◇

Remark 1.3.2. This definition is different from the definition of a DFA in two ways. First, there is a *set* of initial states S, as opposed to a single initial state. Second, the transition function Δ, instead of being a mapping to the set of states Q, is a mapping to the *power set* of states $\wp(Q)$. So, when the transition function Δ is given a state and a symbol, it returns a *set* of states, as opposed to a single state. ◇

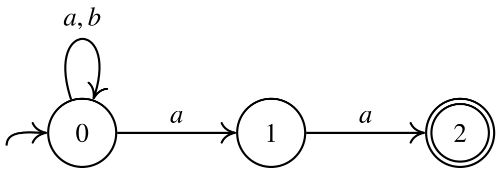
Let us achieve a first understanding of this model by examining some examples.

Example 1.3.3. As an initial example, consider the NFA $M = (Q, \Sigma, S, T, \Delta)$ with state set $Q = \{0, 1, 2\}$, input alphabet $\Sigma = \{a, b\}$, initial state set $S = \{0\}$, accept state set $T = \{2\}$, and the following transition function:

Δ	a	b
0	$\{0,1\}$	$\{0\}$
1	$\{2\}$	$\emptyset$
2	$\emptyset$	$\emptyset$

Observe that the sets $\Delta(2,a)$, $\Delta(1,b)$, and $\Delta(2,b)$ are empty; that each of the sets $\Delta(1,a)$ and $\Delta(0,b)$ contains one element; and that the set $\Delta(0,a)$ contains two elements.

The following is a diagram for this NFA, drawn under the conventions we will use:



We form the diagram for an NFA in the following fashion. Mimicking our convention for DFA, each state is placed in a circle, each initial state is indicated by an unlabeled arrow, and each accept state is in a double circle. For each state p, each symbol c, and each state $q \in \Delta(p,c)$, the diagram includes an arrow from p to q with label c; as with DFA, multiple labels are placed on a single arrow. Observe that the emptiness of such a set $\Delta(p,c)$ translates to the diagram lacking an arrow coming out of the state p with label c.

Let us consider how this example NFA processes the string *baa*. It starts in state 0, which is the only initial state. After reading the symbol b, the NFA can only transition to state 0, as this state is the lone element in $\Delta(0,b) = \{0\}$. From state 0, after reading in the next symbol, a, the NFA may transition to either state 0 or 1, as $\Delta(0,a) = \{0,1\}$. After reading in the last symbol a, from state 0, the NFA may transition to either state 0 or 1; from state 1, the NFA may transition to state 2, the lone element in $\Delta(2,a) = \{2\}$. Hence, after reading in *baa*, the NFA may be in state 0, 1, or 2.

As for a DFA, a *configuration* of an NFA is a state paired with a string. While we will formalize the notion of *successor configuration* of an NFA below, we now look at some examples. As is consistent with the notation for DFA, the symbol $\vdash$ is used between two configurations to indicate that the configuration coming after the symbol is a sucessor configuration of the configuration before the symbol. The following diagram shows all of the configurations reachable when this example NFA is invoked on the string *baa*:

$$\begin{tikzcd}
 & & & {[0,\epsilon]} \\
 & & {[0,a]} \arrow[ur, phantom, sloped, "\vdash"] \arrow[dr, phantom, sloped, "\vdash"] & \\
 & & & {[1,\epsilon]} \\
{[0,baa]} \arrow[r, phantom, "\vdash"] & {[0,aa]} \arrow[uur, phantom, sloped, "\vdash"] \arrow[ddr, phantom, sloped, "\vdash"] & & \\
 & & & \\
 & & {[1,a]} \arrow[r, phantom, "\vdash"] & {[2,\epsilon]}
\end{tikzcd}$$

How does an NFA cast a judgment of acceptance or rejection on a string? After fully reading in the string *baa*, the example NFA M may be in any of its three states. Exactly one of these three states, the state 2, is an accept state. A staunch advocate of democracy might suggest that *baa* ought to be considered rejected, since the majority of these three states are not accept states! Indeed, the fact that an NFA configuration can admit multiple successor configurations may invite the idea of making transitions based on chance. However, for an NFA, acceptance is defined in terms of *possibility*; no real notions of *probability* come into play. In the case of the NFA M, a string is considered accepted when, starting from the initial state 0, there *exists* a choice of transitions such that the NFA terminates in an accept state. Thus, the string *baa* is regarded as accepted by the NFA M.

Next, let us consider this NFA's behavior on the input string *aba*. The following diagram shows the reachable configurations:

$$\begin{tikzcd}
 & & & {[0,\epsilon]} \\
 & {[0,ba]} \arrow[r, phantom, "\vdash"] & {[0,a]} \arrow[ur, phantom, sloped, "\vdash"] \arrow[dr, phantom, sloped, "\vdash"] & \\
{[0,aba]} \arrow[ur, phantom, sloped, "\vdash"] \arrow[dr, phantom, sloped, "\vdash"] & & & {[1,\epsilon]} \\
 & {[1,ba]} & &
\end{tikzcd}$$

Note that the configuration $[1, ba]$ has no successor configurations, since $\Delta(1, b)$ is empty; from this configuration, the computation simply terminates. After processing the entire string *aba*, the NFA M may be in either state 0 or state 1. As neither of these states are accept states, the string *aba* is regarded as rejected.

Which strings are accepted by this NFA? The only way to transition to the accept state 2 is to read an a from state 1, and the only way to transition to state 1 is to read an a from state 0. On the other hand, state 0 also permits transitions to itself, on each of the symbols a and b. Clearly, each string accepted by this NFA must end with *aa*; moreover, any string that ends with *aa* is accepted by this NFA, for the symbols prior to the final *aa* can be processed by staying in state 0, and then the final *aa* can be processed by moving from state 0 to state 2. Hence, the NFA accepts precisely those strings that end with *aa*.

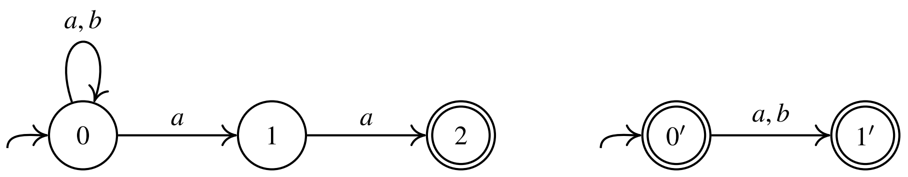
It may be instructive to compare this NFA with the DFA of Example 1.1.5. That DFA's language is the complement of the language accepted by this NFA, but if we modify that DFA so that 2 is its only accept state, its language becomes equal to that of this NFA. This NFA offers a dash of expressional economy over the modified DFA: its diagram contains only 4 arrow labels, in contrast to the DFA diagram's 6 arrow labels! ◇

The computation of a DFA on a string may be said to proceed in a *linear* fashion: each configuration either has a unique successor configuration, or no successor configuration; and the reachable configurations naturally form a linear sequence. On the other hand, the computation of an NFA on a string may be said to proceed in a *branching* fashion: a configuration may have multiple successor configurations; and the reachable configurations naturally form a tree, as seen in Example 1.3.3.

Example 1.3.4. We consider a second NFA $M' = (Q', \Sigma, S', T', \Delta')$, which is an extension of the example NFA M from Example 1.3.3. The NFA M' has state set $Q' = \{0, 1, 2, 0', 1'\}$, input alphabet $\Sigma = \{a, b\}$, initial state set $S' = \{0, 0'\}$, accept state set $T' = \{2, 0', 1'\}$, and the following transition function:

Δ'	a	b
0	$\{0, 1\}$	$\{0\}$
1	$\{2\}$	$\emptyset$
2	$\emptyset$	$\emptyset$
$0'$	$\{1'\}$	$\{1'\}$
$1'$	$\emptyset$	$\emptyset$

The following is a diagram for the NFA M':



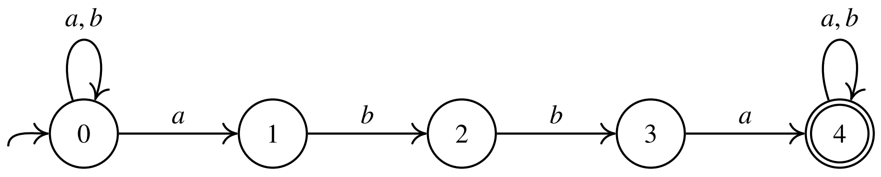
The NFA M' has two initial states, 0 and $0'$. On an input string, an NFA may begin in any of its initial states; if there *exist* a choice of initial state and a choice of transitions from this initial state to one of the accept states, the string is regarded as accepted. For example, the NFA M' accepts the strings a and b, since from the initial state $0'$, both the symbols a and b permit transitions to the state $1'$, which is an accept state. The NFA M' also accepts the empty string ϵ: on this string, it may begin and terminate in the state $0'$, which is both an initial state and an accept state.

It can be seen that when the NFA M' begins in the state $0'$, the strings that can lead to acceptance are precisely ϵ, a, and b. On the other hand, when this NFA begins in the state 0, the strings that can lead to acceptance are exactly the strings accepted by the NFA M of the previous example. Hence, the set of strings accepted by the NFA M' is equal to the union of the set of strings $\{\epsilon, a, b\}$ with the set of strings ending with aa. ◇

Example 1.3.5. We next consider an example NFA that performs a type of substring search. This example NFA has state set $Q = \{0, 1, 2, 3, 4\}$, input alphabet $\Sigma = \{a, b\}$, initial state set $S = \{0\}$, accepting state set $T = \{4\}$, and the following transition function:

Δ	a	b
0	$\{0, 1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\{4\}$	$\emptyset$
4	$\{4\}$	$\{4\}$

The following is a diagram for this example NFA:

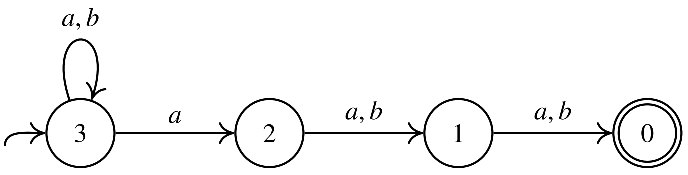


This NFA always begins in state 0, its unique initial state. From that state, the NFA may consume either of the symbols a and b and remain in that state; or it may proceed to state 1 upon reading an a. Once it proceeds to state 1, however, for the computation to stay alive, it must read the symbols b, b, and a, in order, after which it reaches state 4, the only accept state. In state 4, the NFA may consume either of the symbols a and b and remain in that state. From this description, it can be seen that this NFA accepts exactly those strings that contain *abba* as a substring. ◇

Example 1.3.6. We present our final example of an NFA; whether a string is accepted by this NFA depends on the contents of the end of the string, in particular, on the last few symbols in the string (should they exist). This NFA has state set $Q = \{3, 2, 1, 0\}$, input alphabet $\Sigma = \{a, b\}$, initial state set $S = \{3\}$, accept state set $T = \{0\}$, and the following transition table:

Δ	a	b
3	$\{3, 2\}$	$\{3\}$
2	$\{1\}$	$\{1\}$
1	$\{0\}$	$\{0\}$
0	$\emptyset$	$\emptyset$

The following is a diagram for this NFA:



This NFA begins in state 3, its unique initial state. From this state, the NFA may consume any number of a's and b's and remain in this state; the only way to exit this state is to consume an a and move to state 2. From state 2, either symbol permits a unique transition, which is to state 1; from state 1, either symbol permits a unique transition, which is to state 0. While state 0 is accepting, it permits no transitions. Hence, a computation that terminates in state 0 must be timed properly, so that when reading a string, the moment of departure from state 3 allows the coincidence of reaching state 0 and of having scanned the whole string. This coincidence occurs when the a symbol used to transition from state 3 to state 2 is followed by exactly two symbols. We can thus see that a string is accepted by this NFA if and only if it contains three or more symbols, and its third symbol from the right is an a.

The following diagram shows the reachable configurations when this NFA is invoked on the input string aba:

$$\begin{tikzcd}
 & & & {[3,\epsilon]} \\
 & {[3,ba]} \arrow[r, phantom, "\vdash"] & {[3,a]} \arrow[ur, phantom, "\vdash"] \arrow[dr, phantom, "\vdash"] & \\
{[3,aba]} \arrow[ur, phantom, "\vdash"] \arrow[dr, phantom, "\vdash"] & & & {[2,\epsilon]} \\
 & {[2,ba]} \arrow[r, phantom, "\vdash"] & {[1,a]} \arrow[r, phantom, "\vdash"] & {[0,\epsilon]}
\end{tikzcd}$$

◇

Remark 1.3.7. For each of the last two example NFA, the reader is invited to ponder how to construct a DFA sharing the NFA's language, and how many states are needed to construct such a DFA. ◇

We next formalize the notions needed to precisely discuss the behavior of an NFA.

Definition 1.3.8. Let $M = (Q, \Sigma, S, T, \Delta)$ be an NFA.

- A **configuration** of M is a pair $[q, y]$ consisting of a state $q \in Q$ and a string $y \in \Sigma^*$.
- An **initial configuration** of M on a string $y \in \Sigma^*$ is a configuration of the form $[s, y]$, where $s \in S$.
- A configuration $[r, x]$ of M is a **successor configuration** of a configuration $[q, y]$ of M if there exists $a \in \Sigma$ such that $y = ax$ and $r \in \Delta(q, a)$.

To discuss configurations of M, we use the relations presented in Definition 1.1.8. $\diamond$

When dealing with an NFA, observe that in order for a configuration $[q, x]$ to have a successor configuration, it must hold that $|x| \geq 1$. On the other hand, even when $|x| \geq 1$, it is not necessary for a configuration $[q, x]$ to have a successor configuration: the set $\Delta(q, a)$ may be empty, where a denotes the leftmost symbol of x.

In general, when dealing with a computational machine such as a DFA or an NFA, we use the term **computation** to refer to a sequence of configurations that begins with an initial configuration, and where each configuration γ in the sequence is followed by a successor configuration of γ, so long as such a successor configuration exists. When invoked on a string, it is possible for an NFA to carry out multiple computations, as the next example will discuss. Say that a computation is *accepting* if it ends with a configuration whose state is an accept state; under this terminology, an NFA M accepts a string x when there *exists* an accepting computation beginning with the initial configuration of M on x.

Example 1.3.9. Let us revisit Example 1.3.3, and consider its NFA M and its first diagram of configurations. The configuration $[0, aa]$ has two successor configurations, $[0, a]$ and $[1, a]$; we can notate this by writing

$$[0, aa] \vdash_M [0, a] \quad \text{and} \quad [0, aa] \vdash_M [1, a].$$

As $[0, aa]$ is a successor configuration of $[0, baa]$, we can write $[0, baa] \vdash_M [0, aa]$; it then follows that

$$[0, baa] \vdash_M^2 [0, a] \quad \text{and} \quad [0, baa] \vdash_M^2 [1, a].$$

As $[0, a] \vdash_M [0, \epsilon]$, $[0, a] \vdash_M [1, \epsilon]$, and $[1, a] \vdash_M [2, \epsilon]$, we may write

$$[0, baa] \vdash_M^3 [0, \epsilon], \quad [0, baa] \vdash_M^3 [1, \epsilon], \quad \text{and} \quad [0, baa] \vdash_M^3 [2, \epsilon].$$

To unpack and expand the last three relationships shown, we have the following three computations of M that begin with the initial configuration $[0, baa]$:

$$[0, baa] \vdash_M [0, aa] \vdash_M [0, a] \vdash_M [0, \epsilon],$$

$$[0, baa] \vdash_M [0, aa] \vdash_M [0, a] \vdash_M [1, \epsilon],$$

$$[0, baa] \vdash_M [0, aa] \vdash_M [1, a] \vdash_M [2, \epsilon].$$

By glancing back at the first diagram in Example 1.3.3, we can see that there are no further computations of M beginning with the configuration $[0, baa]$. $\diamond$

For an NFA $M = (Q, \Sigma, S, T, \Delta)$, we officially define acceptance and rejection of strings as follows.

Definition 1.3.10. Let $y \in \Sigma^*$ be a string. If there exist states $s \in S$ and $t \in T$ such that $[s, y] \vdash_M^* [t, \epsilon]$, then we say that M **accepts** y; otherwise, we say that M **rejects** y. $\diamond$

Definition 1.3.11. We define the **language** of an NFA M, denoted by $L(M)$, to be the set $\{y \in \Sigma^* \mid M \text{ accepts } y\}$. $\diamond$

Example 1.3.12. Let us continue the discussion in Example 1.3.9. For the NFA M under examination, we had $[0, baa] \vdash_M^3 [2, \epsilon]$; since $0 \in S$ and $2 \in T$, we obtain that the NFA M accepts the string *baa*, via Definition 1.3.10. ◇

We have arrived at a critical juncture in our study. We have, in our hands, *two* computational models, the DFA and the NFA. A natural question that one can pose at this point is how we can compare these two models. Above all, we are interested in what our computational models can *do*, that is, in the languages that they can compute. We thus adopt a functional viewpoint and compare our computational models *externally*, by grading each model according to the span of languages that it defines. Indeed, in general it is not altogether clear how one would compare models *internally*: the mechanisms by which one model computes may be quite qualitatively different from those of another model.

We provide our first such model comparison result by arguing the quite plausible result that each language definable by a DFA is also definable by an NFA. This result reveals that, from the external viewpoint that we adopt, the NFA model is at least as powerful as the DFA model. Before providing the argument, let us emphasize that—strictly speaking—a DFA is not an NFA (nor is an NFA a DFA): as noted in Remark 1.3.2, the definitions of NFA and DFA differ in two parts.

Proposition 1.3.13. *For each DFA M, there exists an NFA M' such that $L(M') = L(M)$.*

This proposition can be argued as follows. Let $M = (Q, \Sigma, s, T, \delta)$ be a DFA. Based on this DFA M, we define the NFA $M' = (Q, \Sigma, S', T, \Delta)$ to have initial state set $S' = \{s\}$ and transition function defined by $\Delta(q, a) = \{\delta(q, a)\}$, for each pair $(q, a) \in Q \times \Sigma$. Since M and M' have the same state set, they have the same configurations. We have that the only start state of the NFA M' is the start state of the DFA M, and that when given any state q and symbol a, the NFA M' has exactly one state to which it can transition—namely, the state to which the DFA M transitions. Indeed, if we were to draw both M and M' as diagrams, the results would be identical. Consequently, the notions of successor configuration coincide for M and M', that is, for all configurations α and β of these automata, β is the successor configuration of α according to M (under Definition 1.1.6) if and only if β is a successor configuration of α according to M' (under Definition 1.3.8). To state this symbolically: for all configurations α and β, it holds that $\alpha \vdash_M \beta$ if and only if $\alpha \vdash_{M'} \beta$. It follows that a string is accepted by M if and only if it is accepted by M'; so, the proposition is established.

We will later prove results that imply that one can convert in the other direction, namely, that for each NFA, there exists a DFA having the same language as the NFA (Theorem 1.3.24). Together, the two conversions imply that the classes of languages induced by each of these two models are equal. As a consequence, to show that there exists a DFA for a given language (that is, that a language is regular), it suffices to show that there exists an NFA for the language. This consequence implies a form of programming convenience: for a given language, it may be easier to present an NFA for the language than to present a DFA for the language.

1.3.2 ϵ-NFA

We next present an extension of the NFA model, called the *ϵ-NFA*. We will prove that each ϵ-NFA has a language that is regular, and hence, the ϵ-NFA model provides yet more convenience for establishing the regularity of a language. It will also be useful for showing further closure properties of the regular languages.

The definition of this extended model is built on the definition of NFA, but has the supplementary feature that, from any state of an ϵ-NFA, additional transitions are permitted. These additional transitions are referred to as **ϵ-transitions**; the transition function Δ of an ϵ-NFA specifies the ϵ-transitions by providing, for each state q, a set of states $\Delta(q, \epsilon)$. Operationally, an ϵ-NFA may, at any point in time, freely make a transition from a state q to a state in the set $\Delta(q, \epsilon)$, without consuming any input symbols.

Definition 1.3.14. An **ϵ-NFA** is a 5-tuple $M = (Q, \Sigma, S, T, \Delta)$ where each of the parts is defined as in the definition of NFA (Definition 1.3.1), except the *transition function* is a mapping

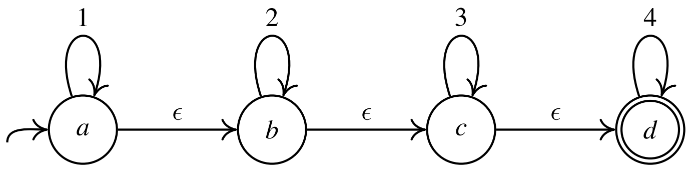
$$\Delta\colon Q \times (\Sigma \cup \{\epsilon\}) \to \wp(Q). \qquad \diamond$$

Recall that $\epsilon \notin \Sigma$ is always assumed.

Example 1.3.15. We present an example ϵ-NFA. In contrast to many of the automata examples given so far, the state set consists of letters, and the input alphabet consists of numbers. The example $N = (Q, \Sigma, S, T, \Delta)$ has state set $Q = \{a, b, c, d\}$, input alphabet $\Sigma = \{1, 2, 3, 4\}$, initial state set $S = \{a\}$, accept state set $T = \{d\}$, and the following transition function:

Δ	1	2	3	4	ϵ
a	$\{a\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{b\}$
b	$\emptyset$	$\{b\}$	$\emptyset$	$\emptyset$	$\{c\}$
c	$\emptyset$	$\emptyset$	$\{c\}$	$\emptyset$	$\{d\}$
d	$\emptyset$	$\emptyset$	$\emptyset$	$\{d\}$	$\emptyset$

We form the diagram for an ϵ-NFA much in the same way that we formed the diagram for an NFA. The only difference is that we include the ϵ-transitions, so whenever $q \in \Delta(p, i)$, we include an arrow from state p to state q with label i; here, we perform this over each element $i \in \Sigma \cup \{\epsilon\}$, that is, including the case that $i = \epsilon$. The following is a diagram for our example ϵ-NFA N:



As with DFA and NFA, a *configuration* of an ϵ-NFA consists of a state paired with a string. We fully formalize the behavior of an ϵ-NFA below; as an example, let us examine the configurations reachable when this automaton is invoked on the string 13:

$$\begin{tikzcd}
 & & & & {[c,\epsilon]} \arrow[r, phantom, "\vdash"] & {[d,\epsilon]} \\
 & {[a,3]} \arrow[r, phantom, "\vdash"] & {[b,3]} \arrow[r, phantom, "\vdash"] & {[c,3]} \arrow[ur, phantom, sloped, "\vdash"] \arrow[dr, phantom, sloped, "\vdash"] & & \\
 & & & & {[d,3]} & \\
{[a,13]} \arrow[uur, phantom, sloped, "\vdash"] \arrow[dr, phantom, sloped, "\vdash"] & & & & & \\
 & {[b,13]} \arrow[r, phantom, "\vdash"] & {[c,13]} \arrow[r, phantom, "\vdash"] & {[d,13]} & &
\end{tikzcd}$$

Let us see cases of how this example ϵ-NFA makes use of ϵ-transitions. Due to the inclusion $b \in \Delta(a, \epsilon)$, from state a the automaton may freely transition to state b without consuming any symbol. This is evidenced in the diagram, where we see the relationships $[a, 13] \vdash [b, 13]$, and $[a, 3] \vdash [b, 3]$. Indeed, most of the transitions shown in the diagram can be recognized to be ϵ-transitions.

It can be seen from the diagram of this automaton that, from the initial state a, the accept state d can only be reached by traversing the states a, b, c, and d in order. In the state a, the automaton may consume the symbol 1 and remain in this state, or it may freely progress to the next state in the order. The states b and c behave similarly, but with respect to the symbols 2 and 3. In state d, the automaton may consume the symbol 4 and remain in this state. From this description, it can be seen that a string is accepted by the automaton if and only if it is *sorted* in the sense that, whenever $i, j \in \Sigma$ are such that $i < j$, each occurrence of i appears before each occurrence of j. ◇

We next formalize the behavior of an ϵ-NFA; the difference with the formalization of NFA is that we extend the definition of *successor configuration* to account for ϵ-transitions.

Definition 1.3.16. Let $M = (Q, \Sigma, S, T, \Delta)$ be an ϵ-NFA.

- A **configuration** of M is a pair $[q, y]$ consisting of a state $q \in Q$ and a string $y \in \Sigma^*$.
- An **initial configuration** of M on a string $y \in \Sigma^*$ is a configuration of the form $[s, y]$, where $s \in S$.
- A configuration $[r, x]$ of M is a **successor configuration** of a configuration $[q, y]$ of M if there exists $a \in \Sigma \cup \{\epsilon\}$ such that $y = ax$ and $r \in \Delta(q, a)$.

To discuss configurations of M, we use the relations presented in Definition 1.1.8. To define the notions of acceptance and rejection for M, we put into effect Definition 1.3.10. ◇

So, as was the case for an NFA, an ϵ-NFA M accepts a string x when there *exists* a computation that begins with an initial configuration of M on x, and that is *accepting* in the sense of ending with a configuration having an accept state.

Definition 1.3.17. We define the **language** of an ϵ-NFA M, denoted by $L(M)$, to be the set $\{y \in \Sigma^* \mid M \text{ accepts } y\}$. ◇

We next show how to convert from an NFA to an ϵ-NFA having the same language. This shows that, in the sense made precise, the ϵ-NFA model is at least as powerful as the NFA model. Recall that we previously showed that the NFA model is at least as powerful as the DFA model (Proposition 1.3.13); the present result is in the spirit of that previous result in that it also compares two computational models.

Proposition 1.3.18. *For each NFA M, there exists an ϵ-NFA M' such that $L(M') = L(M)$.*

We can argue this proposition by designing the ϵ-NFA M' to have no ϵ-transitions, and to be otherwise based on the NFA M. To be precise, from an NFA $M = (Q, \Sigma, S, T, \Delta)$, define M' as the ϵ-NFA $(Q, \Sigma, S, T, \Delta')$ where, for each $q \in Q$, we define $\Delta'(q, a)$ as $\Delta(q, a)$ if $a \in \Sigma$, and as the empty set $\emptyset$ if $a = \epsilon$. It is straightforward to verify that M and M' share the same configurations as well as the same notion of successor configuration. The definition of acceptance is the same for both automata and depends only on the set of start states, the set of accept states, and the notion of successor configuration, all three of which are shared in common by M and M'; consequently, a string is accepted by M if and only if it is accepted by M', and we have confirmed the proposition.

At this point, let us identify some facts about the automata models defined so far. Suppose that M is a DFA, NFA, or ϵ-NFA on alphabet Σ, and let $w \in \Sigma^*$ be any string. If one configuration is the successor of another, then adding the string w to the end of each of the configurations does not change the successor relationship; also, if one configuration is the successor of another, then removing the string w from the end of each of the configurations, when it is possible to do so, does not change the successor relationship. We formally state these two facts as follows. First, if it holds that $[r, x]$ is a successor configuration of a configuration $[q, y]$, that is, $[q, y] \vdash_M [r, x]$, then $[q, yw] \vdash_M [r, xw]$. And the converse holds: if, for configurations $[q, y]$ and $[r, x]$, it holds that $[q, yw] \vdash_M [r, xw]$, then $[q, y] \vdash_M [r, x]$. These facts are verified immediately from the definitions of successor configuration. The following proposition is a consequence of these two facts.

Proposition 1.3.19. *Suppose that M is a DFA, NFA, or ϵ-NFA on alphabet Σ, that $w \in \Sigma^*$ is a string, and that $[p, z]$ and $[p', z']$ are configurations of M. Then,*

$$[p, z] \vdash_M^* [p', z'] \quad \textit{if and only if} \quad [p, zw] \vdash_M^* [p', z'w].$$

1.3.3 From ϵ-NFA to DFA

So far, we have seen three computational models and we have established that they increase successively in power: each language definable by a DFA is definable by an NFA, and each language definable by an NFA is definable by an ϵ-NFA. We now close the loop by showing that each language definable by an ϵ-NFA is definable by a DFA, and hence that these three computational models have the same power in that they each define the same class of languages.

Theorem 1.3.20. *For each ϵ-NFA M, there exists a DFA M' such that $L(M') = L(M)$.*

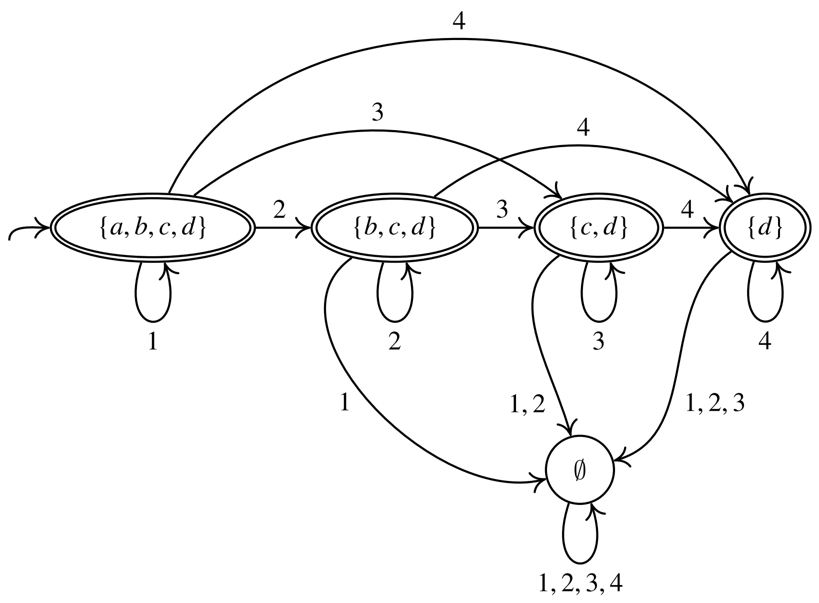


Figure 1.3.1. Part of the DFA $M' = (Q', \Sigma, s', T', \delta)$ obtained by applying the subset construction of Theorem 1.3.20 to the ϵ-NFA $N = (Q, \Sigma, S, T, \Delta)$ of Example 1.3.15. The start state s' contains each state from Q reachable via ϵ-transitions from the start state $a \in S$ of N; each state in Q is reachable in this way, so we have $s' = Q = \{a, b, c, d\}$. The state set Q' of the DFA M' is defined as the power set of Q, but not all states of M' are shown in the diagram; only those reachable from the start state s' via transitions are shown. The ϵ-NFA N has one accept state, d; a state of Q' is an accept state when it contains d. The transition function of the DFA M', given a state U and a symbol a, yields the set of states reachable in N by consuming a along with taking ϵ-transitions, from a state in U. For example, from the state $\{c, d\}$, a transition on 1 yields the empty set: starting from c or d, it is not possible to consume 1 in N, even after making ϵ-transitions. From the state $\{c, d\}$, a transition on 4 yields the set $\{d\}$: in N, from the state d, the symbol 4 can be consumed, but after this no further states can be reached by ϵ-transitions; from the state c, the symbol 4 can be consumed only after making an ϵ-transition to d.

Let $M = (Q, \Sigma, S, T, \Delta)$ be an ϵ-NFA. Our goal is to define a DFA M', based on the ϵ-NFA M, that has the same language as M. One strategy for determining if a string is accepted by an ϵ-NFA is to read in the symbols of the string one-by-one, and to keep in memory *all* of the states that the ϵ-NFA could possibly be in, at each point in time. We show how to construct a DFA M' that, in essence, implements this strategy. The construction of this DFA M' from the ϵ-NFA M is known as the **subset construction**: each state of M' is a subset of the state set of M. An example of this construction is given in Figure 1.3.1.

Define the DFA $M' = (Q', \Sigma, s', T', \delta)$ as follows:

$$\begin{aligned}
Q' &= \wp(Q),\\
s' &= \{u \in Q \mid \exists s \in S \text{ such that } [s, \epsilon] \vdash_M^* [u, \epsilon]\},\\
T' &= \{V \subseteq Q \mid V \cap T \neq \emptyset\},\\
\delta(U, a) &= \{v \in Q \mid \exists u \in U \text{ such that } [u, a] \vdash_M^* [v, \epsilon]\}.
\end{aligned}$$

Let us explain how each of these components is formed.

- This DFA's state set is the power set of Q, since at any point in time, it maintains the set of states from Q that the ϵ-NFA could be in.
- The DFA's start state s' is the set that contains all states reachable from S via ϵ-transitions, in the ϵ-NFA; this set contains S itself, and is the set of all states that the ϵ-NFA could be in prior to reading any symbols. See Figure 1.3.2 for a diagram.
- A state V of the DFA should be regarded as an accept state as long as it contains an accept state of the ϵ-NFA.
- Finally, when U is a state of the DFA and a is a symbol, the state given by the transition function is the set that includes a state v if it is reachable from *some* state in U by consuming a, and possibly allowing ϵ-transitions as well. See Figure 1.3.3 for a diagram.

The following definition is useful for reasoning about our automata.

Definition 1.3.21. Relative to an ϵ-NFA $M = (Q, \Sigma, S, T, \Delta)$, a set $U \subseteq Q$ of states is called **ϵ-closed** when, for each $u \in U$ and each $w \in Q$, if $[u, \epsilon] \vdash_M^* [w, \epsilon]$, then $w \in U$. ◇

So, relative to an ϵ-NFA, a set U of states is ϵ-closed when U contains any state w that is reachable, purely via ϵ-transitions, from a state in U. In reasoning about subsets of Q (that is, sets of states of M), our focus will be on those that are ϵ-closed. The next lemma shows that the set s' has this property.

Lemma 1.3.22. *The set $s' \subseteq Q$ of states is ϵ-closed.*

Proof. Suppose that $u \in s'$ and $w \in Q$ is such that $[u, \epsilon] \vdash_M^* [w, \epsilon]$. By the definition of s', there exists $s \in S$ such that $[s, \epsilon] \vdash_M^* [u, \epsilon]$. It follows that $[s, \epsilon] \vdash_M^* [w, \epsilon]$, and thus by the definition of s', we obtain that $w \in s'$. □

The following lemma relates the transitions of the DFA M' to the transitions of the ϵ-NFA M; in particular, it characterizes the state W that the DFA will be in when it starts in an ϵ-closed state U and reads a string y.

Lemma 1.3.23. *Let $y \in \Sigma^*$ be a string of length n; suppose that $U \subseteq Q$ is ϵ-closed; and let $W \subseteq Q$ be the unique set such that $[U, y] \vdash_{M'}^n [W, \epsilon]$. Then, it holds that $w \in W$ if and only if there exists $u \in U$ such that $[u, y] \vdash_M^* [w, \epsilon]$.*

Proof. We prove this by induction on n.

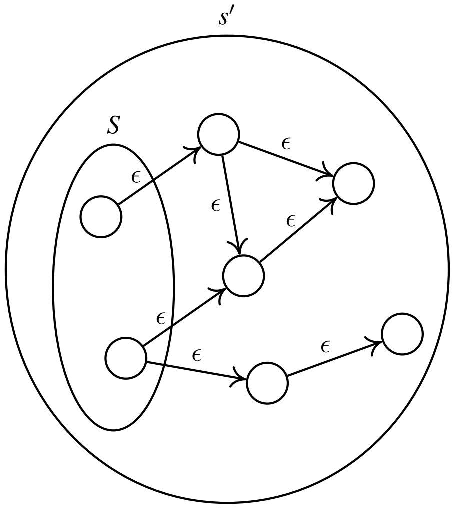


Figure 1.3.2. The start state of the DFA in the subset construction. In the conversion of an ϵ-NFA to a DFA, the start state s' of the DFA is defined as the set of all states reachable from any start state of the ϵ-NFA, by making 0 or more ϵ-transitions. In particular, this start state s' contains the set S of start states of the ϵ-NFA, that is, it holds that $S \subseteq s'$.

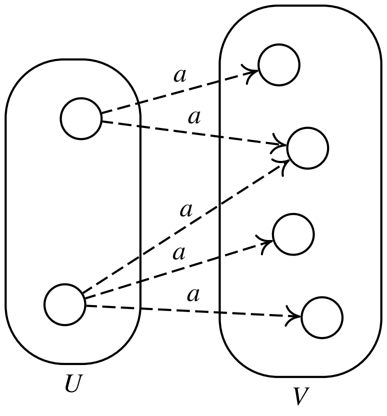


Figure 1.3.3. The transition function of the DFA in the subset construction. In the conversion of an ϵ-NFA to a DFA M', each state of the DFA is a subset of the state set Q of the ϵ-NFA. The transition function of the DFA is defined so that, when given a set $U \subseteq Q$ of ϵ-NFA states along with a symbol a, the function returns the set $V \subseteq Q$ of ϵ-NFA states that can be reached from a state in U after consuming the symbol a. Here, a dotted arrow with label a from a first state to a second state indicates that the second state can be reached from the first by traversing a sequence of states, such that only the symbol a is consumed: thus, in such a traversal, one transition is on the symbol a, and all others are ϵ-transitions.

Suppose that $n = 0$; we then have $y = \epsilon$ and $W = U$. For the forward direction, suppose that $w \in W$; we have $w \in U$ and $[w, y] \vdash_M^* [w, \epsilon]$. For the backward direction, suppose that there exists $u \in U$ such that $[u, y] \vdash_M^* [w, \epsilon]$; then, by the assumption that U is ϵ-closed, it holds that $w \in U = W$.

Suppose that $n > 0$, and write $y = ax$ where $a \in \Sigma$ and $x \in \Sigma^*$. Let $V \subseteq Q$ be the unique set such that $[U, ax] \vdash_{M'} [V, x] \vdash_{M'}^{n-1} [W, \epsilon]$. We have that $V = \delta(U, a)$, implying by the definition of δ that

$$v \in V \quad \Leftrightarrow \quad \text{there exists } u \in U \text{ such that } [u, a] \vdash_M^* [v, \epsilon].$$

The set V is ϵ-closed: suppose that $v \in V$, $w \in Q$, and $[v, \epsilon] \vdash_M^* [w, \epsilon]$; by the above description of V, we have that there exists $u \in U$ such that $[u, a] \vdash_M^* [v, \epsilon]$, implying that $[u, a] \vdash_M^* [w, \epsilon]$, from which it follows that $w \in V$ by the above description of V. By induction, we obtain that

$$w \in W \quad \Leftrightarrow \quad \text{there exists } v \in V \text{ such that } [v, x] \vdash_M^* [w, \epsilon].$$

We now verify each of the two directions of the claim of the lemma.

- For the forward direction, assume that $w \in W$. Then, by the above description of W, there exists $v \in V$ such that $[v, x] \vdash_M^* [w, \epsilon]$. In turn, by the above description of V, there exists $u \in U$ such that $[u, a] \vdash_M^* [v, \epsilon]$, from which it follows that $[u, ax] \vdash_M^* [v, x]$, by Proposition 1.3.19. From the facts that $[u, ax] \vdash_M^* [v, x]$ and $[v, x] \vdash_M^* [w, \epsilon]$, we obtain that $[u, ax] \vdash_M^* [w, \epsilon]$, as desired.
- For the backward direction, assume that there exists $u \in U$ such that $[u, y] \vdash_M^* [w, \epsilon]$; we want to show that $w \in W$. Consider the sequence of configurations that witnesses the relationship $[u, y] \vdash_M^* [w, \epsilon]$: there exist states $q_1, \ldots, q_k, r \in Q$ (with $k \geq 0$) such that

  $$[u, ax] \vdash_M [q_1, ax] \vdash_M \cdots \vdash_M [q_k, ax] \vdash_M [r, x] \vdash_M^* [w, \epsilon].$$

  From this, we obtain $[u, a] \vdash_M^* [r, \epsilon]$ (via Proposition 1.3.19) and $[r, x] \vdash_M^* [w, \epsilon]$. By the above description of V and the fact that $[u, a] \vdash_M^* [r, \epsilon]$, we obtain that $r \in V$; then, by the above description of W and the fact that $[r, x] \vdash_M^* [w, \epsilon]$, we conclude that $w \in W$. □

By making use of this lemma, we can now establish the theorem.

Proof of Theorem 1.3.20. Let $y \in \Sigma^*$ be a string of length n, and let $W \subseteq Q$ be the unique state of Q' such that $[s', y] \vdash_{M'}^n [W, \epsilon]$. By Lemma 1.3.22, the set $s' \subseteq Q$ is ϵ-closed. By Lemma 1.3.23, we obtain that

$$w \in W \quad \Leftrightarrow \quad \text{there exists } u \in s' \text{ such that } [u, y] \vdash_M^* [w, \epsilon].$$

We argue that the ϵ-NFA M accepts y if and only if the DFA M' accepts y.

Suppose that the ϵ-NFA M accepts y. Then there exist states $s \in S$ and $t \in T$ giving the relationship $[s, y] \vdash_M^* [t, \epsilon]$. Since $s \in s'$, by the above characterization of W, we have the inclusion $t \in W$. This implies that $W \cap T \neq \emptyset$, so $W \in T'$ and the DFA M' accepts y.

Suppose that the DFA M' accepts y. Then $W \in T'$, implying that $W \cap T \neq \emptyset$. Let t be a state (of Q) that is in $W \cap T$. By the above characterization of W, there exists $u \in s'$ such that $[u, y] \vdash_M^* [t, \epsilon]$. By definition of s', there exists $s \in S$ such that $[s, \epsilon] \vdash_M^* [u, \epsilon]$, implying that $[s, y] \vdash_M^* [u, y]$. From the results $[s, y] \vdash_M^* [u, y]$ and $[u, y] \vdash_M^* [t, \epsilon]$, it follows that $[s, y] \vdash_M^* [t, \epsilon]$. Since $s \in S$ and $t \in T$, we obtain that M accepts y. □

1.3.4 Summary

The following theorem results from collecting together our comparisons between automata models.

Theorem 1.3.24. *Let B be a language. The following are equivalent:*

- *There exists a DFA M such that $L(M) = B$; that is, B is regular.*
- *There exists an NFA M such that $L(M) = B$.*
- *There exists an ϵ-NFA M such that $L(M) = B$.*

Proof. If there exists a DFA whose language is B, then by Proposition 1.3.13, there exists an NFA whose language is B. If there exists an NFA whose language is B, then by Proposition 1.3.18, there exists an ϵ-NFA whose language is B. And if there exists an ϵ-NFA whose language is B, then by Theorem 1.3.20, there exists a DFA whose language is B. □

1.4 More closure properties

In this section, we show that the regular languages enjoy two further closure properties. These results help us understand further the extent of the regular languages, and provide insight into what types of languages can be shown to be regular. These results will also have starring roles in the next section, where we will see that the regular languages can in fact be *characterized* using natural closure properties. In the present section, to establish each of the two closure properties under scrutiny, we build an ϵ-NFA whose language is the language claimed to be regular; hence, we rely crucially on the just-established fact that ϵ-NFA define regular languages (this fact follows from Theorem 1.3.24). Let us remark that this fact also allows for an alternative proof that the regular languages are closed under union: given two ϵ-NFA M_B, M_C, one can build an ϵ-NFA whose language is the union $L(M_B) \cup L(M_C)$ essentially by drawing the diagrams of M_B and M_C side by side, and interpreting the overall result as the diagram of an ϵ-NFA.

1.4.1 Concatenation

Let B and C be languages. The **concatenation** of B and C, denoted by $B \cdot C$ or by BC, is defined as $\{xy \mid x \in B \text{ and } y \in C\}$, that is, as the set containing each string that can be obtained by concatenating a string in B with a string in C.

Theorem 1.4.1. *If B and C are both regular languages over the same alphabet Σ, then their concatenation $B \cdot C$ is also a regular language.*

Let $M_B = (Q_B, \Sigma, S_B, T_B, \Delta_B)$ and $M_C = (Q_C, \Sigma, S_C, T_C, \Delta_C)$ be ϵ-NFA with $L(M_B) = B$ and $L(M_C) = C$. We assume that the state sets Q_B and Q_C are disjoint. (If they are not disjoint, the states in one of the sets may be renamed in order to achieve disjointness, without affecting the language of its automaton; Remark 1.1.4 discussed state renaming in DFA, and applies equally well to NFA and ϵ-NFA.) From these two ϵ-NFA, we build a third ϵ-NFA M, whose language is the concatenation $B \cdot C$. On a high level, this is done by including all of the states and transitions in both M_B and M_C, designating the states in S_B as the initial states and the states in T_C as the accept states, and adding ϵ-transitions from each state in T_B to each state in S_C. Figure 1.4.1 gives a diagram indicating this construction.

Essentially, a string will be accepted by M if and only if it can be split into two parts, where the first part allows for M to move from a state in S_B to a state in T_B (that is, the first part is accepted by M_B), and the second part allows for M to move from a state in S_C to a state in T_C (that is, the second part is accepted by M_C). The added ϵ-transitions allow for free passage from T_B to S_C, and are M's only transitions linking the two original ϵ-NFA.

Formally, we define the ϵ-NFA M as $(Q_B \cup Q_C, \Sigma, S_B, T_C, \Delta)$ where the transition function Δ is defined as follows:

- For all $q \in Q_B$ and $a \in \Sigma \cup \{\epsilon\}$, define $\Delta(q, a)$ as $\Delta_B(q, a) \cup S_C$ if $q \in T_B$ and $a = \epsilon$, and as $\Delta_B(q, a)$ otherwise.
- For all $q \in Q_C$ and $a \in \Sigma \cup \{\epsilon\}$, define $\Delta(q, a)$ as $\Delta_C(q, a)$.

So, the definition of Δ naturally imitates those of Δ_B and Δ_C, but in the particular case of a state $q \in T_B$ and the symbol $a = \epsilon$, transitions to the states in S_C are allowed in addition to the transitions given by Δ_B.

Proof of Theorem 1.4.1. We prove that the ϵ-NFA M just defined has $L(M) = B \cdot C$; this suffices by Theorem 1.3.24. We first establish the containment $L(M) \supseteq B \cdot C$, which is readily done. We then establish the containment $L(M) \subseteq B \cdot C$, which involves analyzing the transitions made by an accepting computation of M, and showing that the string accepted can be split into two parts, where the first is accepted by M_B, and the second is accepted by M_C. This latter part of the proof is a bit tedious, but relatively straightforward.

Suppose that $z \in B \cdot C$. Then there exist strings $x \in B$, $y \in C$ such that $z = xy$. Due to the inclusions $x \in L(M_B)$ and $y \in L(M_C)$, there exist states $s_B \in S_B$ and $t_B \in T_B$ such that $[s_B, x] \vdash^*_{M_B} [t_B, \epsilon]$; and there exist $s_C \in S_C$ and $t_C \in T_C$ such that $[s_C, y] \vdash^*_{M_C} [t_C, \epsilon]$.

- From the definition of Δ, we have that $[s_B, x] \vdash^*_M [t_B, \epsilon]$ and $[s_C, y] \vdash^*_M [t_C, \epsilon]$.
- It then follows from Proposition 1.3.19 that $[s_B, xy] \vdash^*_M [t_B, y]$.
- By definition of Δ, we have $s_C \in \Delta(t_B, \epsilon)$, implying that $[t_B, y] \vdash_M [s_C, y]$.

Combining $[s_B, xy] \vdash^*_M [t_B, y]$, $[t_B, y] \vdash_M [s_C, y]$, and $[s_C, y] \vdash^*_M [t_C, \epsilon]$, we obtain immediately that $[s_B, xy] \vdash^*_M [t_C, \epsilon]$, implying that $xy \in L(M)$.

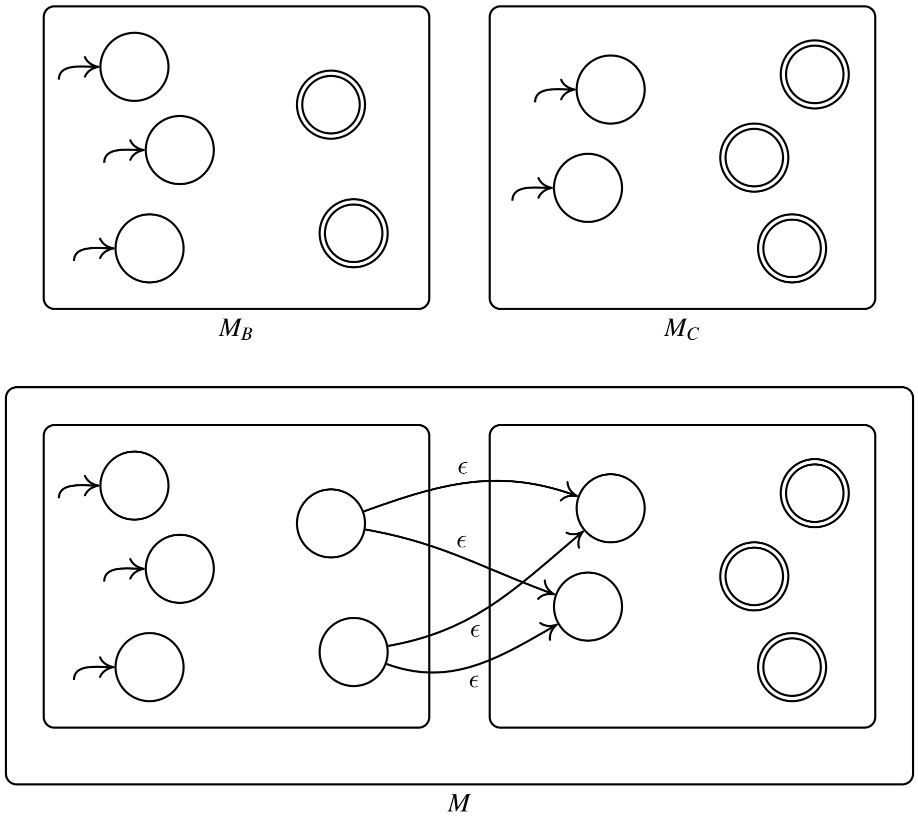


Figure 1.4.1. The construction of Theorem 1.4.1. Given two ϵ-NFA M_B and M_C, a third ϵ-NFA M is formed whose language is equal to the concatenation of the languages $L(M_B)$ and $L(M_C)$. In this diagram, only the start and accept states of M_B and M_C are depicted; the other states and the transitions of these automata are not shown. The ϵ-NFA M is formed by adding ϵ-transitions from the accept states of M_B to the initial states of M_C, by designating the start state set of M to be the start state set of M_B, and by designating the accept state set of M to be the accept state set of M_C.

Suppose that $z \in L(M)$. Then there exist $s_B \in S_B$ and $t_C \in T_C$ such that $[s_B, z] \vdash_M^* [t_C, \epsilon]$. It follows that there exist configurations $\gamma_0, \ldots, \gamma_k$ of M such that $\gamma_0 = [s_B, z]$, $\gamma_k = [t_C, \epsilon]$, and $\gamma_0 \vdash_M \gamma_1 \vdash_M \cdots \vdash_M \gamma_k$. Set q_j to be the state of the configuration γ_j, for each index j.

- Let i be the index such that, in the list $q_0, \ldots, q_k$, it holds that q_i is the first state in Q_C. We have that i is well-defined since $q_k = t_C \in Q_C$; also, as $q_0 = s_B \notin Q_C$, we have the inequality $i > 0$.

- By the definition of Δ, whenever $q \in Q_C$ (and $a \in \Sigma \cup \{\epsilon\}$), the set $\Delta(q, a)$ only contains states in Q_C; it follows that $q_i, q_{i+1}, \ldots, q_k$ are all elements of Q_C, and $q_0, \ldots, q_{i-1}$ are all elements of Q_B.
- The transition $\gamma_{i-1} \vdash_M \gamma_i$ must be witnessed by a symbol a such that $q_i \in \Delta(q_{i-1}, a)$; given that $q_{i-1} \in Q_B$ and $q_i \in Q_C$, the definition of Δ implies that $q_{i-1} \in T_B$, $q_i \in S_C$, and $a = \epsilon$. Hence, there exists a string y such that $\gamma_{i-1} = [q_{i-1}, y]$ and $\gamma_i = [q_i, y]$. So, we have $[s_B, z] \vdash_M^* [q_{i-1}, y] \vdash_M [q_i, y] \vdash_M^* [t_C, \epsilon]$.
- Let x be the string such that $z = xy$. From Proposition 1.3.19, $[s_B, z] \vdash_M^* [q_{i-1}, y]$ implies $[s_B, x] \vdash_M^* [q_{i-1}, \epsilon]$, which in turn implies $[s_B, x] \vdash_{M_B}^* [q_{i-1}, \epsilon]$ (by the definition of Δ); we obtain that $x \in L(M_B)$.
- From $[q_i, y] \vdash_M^* [t_C, \epsilon]$, it follows that $[q_i, y] \vdash_{M_C}^* [t_C, \epsilon]$ (by the definition of Δ); we obtain that $y \in L(M_C)$.

We conclude that $z = xy$, where $x \in B$ and $y \in C$. □

1.4.2 Star

When B is a language, we define B^* as the language

$$\{x_1 \cdots x_k \mid k \geq 0 \text{ and } x_1, \ldots, x_k \in B\};$$

that is, B^* is the language containing each string that is the concatenation of 0 or more strings from B. We sometimes refer to B^* as the **star** of B. In the case that $k = 0$, we understand $x_1 \ldots x_k$ to denote the empty string ϵ; so, for any language B, it holds that $\epsilon \in B^*$. In using Σ^* to denote the set of all strings over an alphabet Σ, we have already made use of this notation; observe that this usage is consistent with and generalized by the given definition of B^* for *any* language B.

Theorem 1.4.2. *If B is a regular language, then B^* is also a regular language.*

Let $M = (Q, \Sigma, S, T, \Delta)$ be an ϵ-NFA with $L(M) = B$. From this ϵ-NFA, we build another ϵ-NFA M' whose language is B^*. On a high level, this is done by starting with the states and transitions of M, and adding a new state p, which has ϵ-transitions *to* the initial states of M, and *from* the accept states of M; this state p is defined to be the sole initial state and the sole accept state of M'. A diagram indicating this construction of M' is given in Figure 1.4.2. Formally, define $M' = (Q \cup \{p\}, \Sigma, \{p\}, \{p\}, \Delta')$ where p is assumed to be a new state not in Q, and Δ' is defined as follows. Set $\Delta'(p, \epsilon) = S$ and, for each $a \in \Sigma$, set $\Delta'(p, a) = \emptyset$. For each $q \in Q$ and $a \in \Sigma \cup \{\epsilon\}$, set $\Delta'(q, a)$ to be $\Delta(q, a) \cup \{p\}$ if $q \in T$ and $a = \epsilon$, and to be $\Delta(q, a)$ otherwise.

Proof. We prove that, for the ϵ-NFA M' just defined, it holds that $L(M') = B^*$; this suffices by Theorem 1.3.24. Paralleling the previous proof, the containment $L(M') \supseteq B^*$ is relatively straightforward to show, whereas showing the containment $L(M') \subseteq B^*$ involves analyzing an arbitrary accepting computation of M', and requires more reasoning.

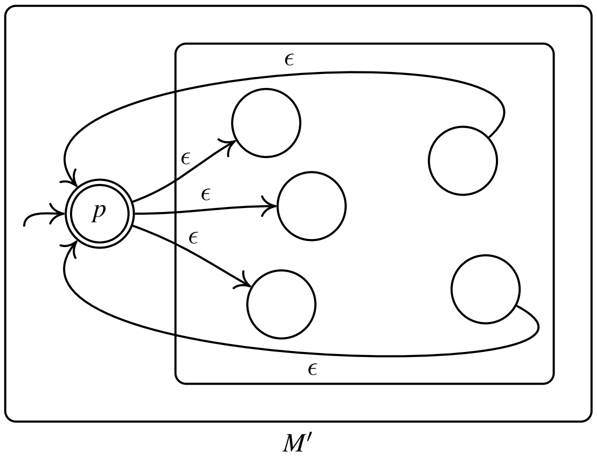


Figure 1.4.2. The construction of Theorem 1.4.2 applied to the ϵ-NFA M_B from Figure 1.4.1. Given an ϵ-NFA M, a second ϵ-NFA M' is formed whose language is equal to the star of M's language. The ϵ-NFA M' is formed by adding a new state p which is both its unique start state and its unique accept state; the state p has ϵ-transitions to each of the initial states of M, and ϵ-transitions from each of the accept states of M.

Suppose that $z \in B^*$. Then there exist strings $x_1, \ldots, x_k \in B$ such that $z = x_1 \cdots x_k$. For each $i = 1, \ldots, k$, there thus exist $s_i \in S$ and $t_i \in T$ such that $[s_i, x_i] \vdash_M^* [t_i, \epsilon]$; it follows from the definition of M' that $[s_i, x_i] \vdash_{M'}^* [t_i, \epsilon]$. From Proposition 1.3.19, we obtain that

$$[s_1, x_1 \cdots x_k] \vdash_{M'}^* [t_1, x_2 \cdots x_k],\ [s_2, x_2 \cdots x_k] \vdash_{M'}^* [t_2, x_3 \cdots x_k],\ \ldots,\ [s_k, x_k] \vdash_{M'}^* [t_k, \epsilon].$$

Using these relationships along with the definition of Δ', we have the following computation of M':

$$\begin{aligned}
[p, x_1 \cdots x_k] &\vdash_{M'} [s_1, x_1 \cdots x_k] \\
&\vdash_{M'}^* [t_1, x_2 \cdots x_k] \vdash_{M'} [p, x_2 \cdots x_k] \vdash_{M'} [s_2, x_2 \cdots x_k] \\
&\vdash_{M'}^* [t_2, x_3 \cdots x_k] \vdash_{M'} [p, x_3 \cdots x_k] \vdash_{M'} [s_3, x_3 \cdots x_k] \\
&\vdots \\
&\vdash_{M'}^* [t_k, \epsilon] \vdash_{M'} [p, \epsilon].
\end{aligned}$$

As $[p, x_1 \cdots x_k] \vdash_{M'}^* [p, \epsilon]$, we obtain $z \in L(M')$.

Suppose that $z \in L(M')$. If $z = \epsilon$, then clearly $z \in B^*$, so assume that $z \neq \epsilon$. Then, there exist configurations $\gamma_0, \ldots, \gamma_n$ such that

$$\gamma_0 = [p, z], \quad \gamma_n = [p, \epsilon], \quad \text{and} \quad \gamma_0 \vdash_{M'} \gamma_1 \vdash_{M'} \cdots \vdash_{M'} \gamma_n.$$

Define the strings $z_1, \ldots, z_\ell$ so that $[p, z_1], \ldots, [p, z_\ell]$ is a list of the configurations from the list $\gamma_0, \ldots, \gamma_n$ that have state p, in order; note that $z_1 = z$, $z_\ell = \epsilon$, and $\ell \geq 2$ (as $z \neq \epsilon$). For each index $i = 1, \ldots, \ell - 1$, it holds that $[p, z_i] \vdash^*_{M'} [p, z_{i+1}]$. Since, according to Δ', the only transitions from p are to states in S, and the only transitions to p are from states in T, there exist states $s_i \in S$, $t_i \in T$ such that

$$[p, z_i] \vdash_{M'} [s_i, z_i] \vdash^*_{M'} [t_i, z_{i+1}] \vdash_{M'} [p, z_{i+1}].$$

By the definition of Δ', it follows that, in the ϵ-NFA M,

$$[s_i, z_i] \vdash^*_M [t_i, z_{i+1}].$$

For each index $i = 1, \ldots, \ell - 1$, let x_i be the string such that $z_i = x_i z_{i+1}$; by Proposition 1.3.19, we have the relationship

$$[s_i, x_i] \vdash^*_M [t_i, \epsilon],$$

implying that $x_i \in L(M)$. We have that the string z can be expanded as

$$z = z_1 = x_1 z_2 = x_1 x_2 z_3 = \cdots = x_1 \cdots x_{\ell-1} z_\ell = x_1 \cdots x_{\ell-1}.$$

Since each of the strings $x_1, \ldots, x_{\ell-1}$ is in $L(M)$, we conclude that $z \in B^*$. □

1.5 Regular expressions

We have studied three computational models, the DFA, the NFA, and the ϵ-NFA. An automaton of any of these three types, as seen, specifies a regular language. In this section, we encounter and study another way, a textual way, of specifying a regular language: giving a *regular expression*, which is a particular type of string. Presenting a regular expression may offer the benefit that there can be a close conceptual correspondence between a regular expression and the language it specifies. Indeed, text searching programs often expect regular expressions as input, although the particular syntax expected may vary.

1.5.1 Definition and evaluation

Definition 1.5.1. Let Σ be an alphabet. We define a **regular expression** over Σ to be a string that can be derived by applying the following rules a finite number of times.

- $\emptyset$ is a regular expression.
- a is a regular expression, for each $a \in \Sigma \cup \{\epsilon\}$.
- $\alpha + \beta$ is a regular expression, when α and β are regular expressions.
- $\alpha\beta$ is a regular expression, when α and β are regular expressions.
- α^* is a regular expression, when α is a regular expression.
- (α) is a regular expression, when α is a regular expression.

Observe that each regular expression over Σ is a string over the alphabet obtained by starting from Σ and adding the symbol $\emptyset$, the symbol ϵ, the symbol $+$, the symbol *, the left parenthesis, and the right parenthesis. Throughout, we assume that none of these additional symbols is contained in any alphabet Σ over which we form regular expressions. ◇

Example 1.5.2. Let Σ be the alphabet $\{a, b\}$. The following are examples of regular expressions over Σ:

$$ab + b^*, \quad ab^*, \quad (ab)^* + b, \quad b^*b + aa(a + \epsilon). \qquad \diamond$$

Let us emphasize that each regular expression is just a string. We will give semantic meaning to the regular expressions by explaining how each regular expression α *evaluates* to a language, denoted by $L(\alpha)$. While we define this evaluation more precisely in the sequel, let us give a preview of how the evaluation is performed. The expression $\emptyset$ evaluates to the language $\emptyset$; each expression $a \in \Sigma \cup \{\epsilon\}$ evaluates to the language $\{a\}$; and the sum of two expressions evaluates to the union of their languages. The next two cases are evaluated in a natural way: the concatenation of two expressions evaluates to the concatenation of the two corresponding languages, and the star of an expression evaluates to the star of its language. Parentheses are used to control the order of evaluation.

Example 1.5.3. We here present some examples of how regular expressions evaluate to languages. Let Σ be the alphabet $\{a, b\}$.

We have $L(a) = \{a\}$ and $L(b) = \{b\}$. Next, consider the regular expression ab; since it is the concatenation of the regular expressions a and b, its language $L(ab)$ is the concatenation of $L(a)$ and $L(b)$, which is the language $\{a\} \cdot \{b\} = \{ab\}$. More generally, if we take any string $x \in \Sigma^*$, it holds that $L(x) = \{x\}$, that is, x as a regular expression evaluates to the language that contains x as its sole element.

Consider now the regular expression $(ab)^*$. As it arises from applying the star to the expression (ab), its language is the star of $L(ab) = \{ab\}$, that is, $L((ab)^*) = \{ab\}^*$. From the definition of the star operator, we know that $\{ab\}^* = \{\epsilon, ab, abab, ababab, \ldots\}$.

Next, consider the regular expression $b + (ab)^*$. This regular expression is the sum of the expressions b and $(ab)^*$, and the language it evaluates to is thus the union of the languages $L(b)$ and $L((ab)^*)$. So,

$$L(b + (ab^*)) = L(b) \cup L((ab)^*) = \{b\} \cup \{\epsilon, ab, abab, ababab, \ldots\}. \qquad \diamond$$

In order to formally define how the evaluation of regular expressions is performed, we first need to discuss the *precedence* of the operations, that is, the order in which the operations are evaluated. The notion of precedence is likely already familiar to the reader: in arithmetic, by a usual convention, division has *higher precedence* than subtraction, so in evaluating an arithmetic expression such as $9 - 6/3$, it is the convention to evaluate the division before the subtraction, and this expression evaluates to $9 - (6/3) = 9 - 2 = 7$. (Note that if the subtraction was evaluated first, the result would be $3/3 = 1$ and hence different.) In evaluating regular expressions, we adhere to the following convention: the star (*) has the highest precedence, followed by concatenation, followed by sum (+). Let us understand what this implies via an example.

Example 1.5.4. In order to determine the language $L(ab + c^*)$ of the expression $ab + c^*$ in accordance with the precedence just given, we first determine $L(c^*)$, next determine $L(ab)$, and then compute $L(ab + c^*)$ as the union of $L(ab)$ and $L(c^*)$. That is, in evaluating the expression $ab + c^*$, we evaluate the + operator last. So, to determine $L(ab + c^*)$, we view $ab + c^*$ as having the form $\alpha + \beta$ where $\alpha = ab$ and $\beta = c^*$; then, $L(\alpha)$ and $L(\beta)$ are evaluated individually, and $L(\alpha + \beta)$ is evaluated as $L(\alpha) \cup L(\beta)$. We have $L(\alpha) = \{ab\}$ and $L(\beta) = \{\epsilon, c, cc, ccc, \dots\}$, so we obtain

$$L(ab + c^*) = \{ab\} \cup \{\epsilon, c, cc, ccc, \dots\}. \qquad \diamond$$

The use of parentheses allows one to control the order of evaluation, in the usual fashion: in any regular expression α, parenthesized expressions occurring within α are evaluated prior to evaluating operators not contained within parentheses. We illustrate this usage via an example.

Example 1.5.5. Consider the expression $(ab + c)^*$, which differs from the expression in the previous example only in that a pair of parentheses has been added. To evaluate this expression, the parenthesized portion would be completely evaluated prior to evaluating the star. The expression $(ab + c)$ evaluates to $L((ab + c)) = \{ab, c\}$. From this, to determine the language $L((ab + c)^*)$, we apply the star to $L((ab + c))$, so

$$L((ab + c)^*) = L((ab + c))^* = \{ab, c\}^*.$$

Let us underscore that this language is different from the language $L(ab + c^*)$; for instance, it holds that $abab \in L((ab + c)^*)$, but $abab \notin L(ab + c^*)$. $\diamond$

The *associativity* of operators is another consideration that ought to be discussed, but turns out to be less important in our present context. In dealing with an expression such as $\epsilon + a + b$, it should technically be specified which of the two + operators should be evaluated first. While the result is independent of the order (as a result of the union $\cup$ being an *associative* operation), we formally adhere to the convention that when dealing with multiple occurrences of +, evaluation proceeds from left to right, so in the example expression, the first instance of + is evaluated prior to the second. The same issue arises in dealing with concatenation (and note that concatenation of languages is also an associative operation); we similarly evaluate multiple occurrences of concatenation from left to right.

We can now officially define the language associated to a regular expression. Let α, β, and γ be regular expressions. We say that γ **has the form** $\alpha + \beta$ if it is syntactically equal to $\alpha + \beta$ and α and β are evaluated prior to the sum indicated by the +, that is, the + is the last operator to be evaluated, according to the presented precedence and associativity. In a similar fashion, we say that γ **has the form** $\alpha\beta$ if it is syntactically equal to $\alpha\beta$ and α and β are evaluated prior to the concatenation; and we say that γ **has the form** α^* if it is syntactically equal to α^* and α is evaluated prior to the star. Lastly, we say that γ **has the form** (α) simply if it is syntactically equal to (α).

Definition 1.5.6. For each regular expression γ over an alphabet Σ, the language $L(\gamma)$ is defined inductively, as follows.

- $L(\emptyset) = \emptyset$.
- $L(a) = \{a\}$, for each $a \in \Sigma \cup \{\epsilon\}$.
- $L(\gamma) = L(\alpha) \cup L(\beta)$, when γ has the form $\alpha + \beta$.
- $L(\gamma) = L(\alpha) \cdot L(\beta)$, when γ has the form $\alpha\beta$.
- $L(\gamma) = L(\alpha)^*$, when γ has the form α^*.
- $L(\gamma) = L(\alpha)$, when γ has the form (α). ◇

Example 1.5.7. As an example, let us consider how to give a regular expression for the language D defined to contain all *alternating strings* over $\Sigma = \{a, b\}$. We here define an **alternating string** to be a string where each occurrence of a, if followed at all, is followed by a b; and where each occurrence of b, if followed at all, is followed by an a. Alternatively, we could say that an alternating string is a string that does not contain two consecutive occurrences of the same symbol.

Consider the regular expressions $(ab)^*$ and $(ba)^*$. Certainly, all of the strings in the languages $L((ab)^*) = \{\epsilon, ab, abab, \dots\}$ and $L((ba)^*) = \{\epsilon, ba, baba, \dots\}$ are alternating. However, these languages do not cover all alternating strings, since they do not include any alternating strings of odd length. We may obtain the alternating strings of odd length by considering the regular expressions $a(ba)^*$ and $b(ab)^*$, whose languages are $L(a(ba)^*) = \{a, aba, ababa, \dots\}$ and $L(b(ab)^*) = \{b, bab, babab, \dots\}$. Putting things together, we have the following expression δ_1 where $L(\delta_1) = D$:

$$\delta_1 = (ab)^* + (ba)^* + a(ba)^* + b(ab)^*.$$

Observe that the regular expressions $(ab)^*a$ and $(ba)^*b$ yield the same languages as $a(ba)^*$ and $b(ab)^*$, respectively: $L((ab)^*a) = L(a(ba)^*)$ and $L((ba)^*b) = L(b(ab)^*)$. Consequently, we arrive at a second expression δ_2 with $L(\delta_2) = D$:

$$\delta_2 = (ab)^* + (ba)^* + (ab)^*a + (ba)^*b.$$

We may develop another regular expression for D that only uses one instance of the star, as follows. Begin with the expressions $(ab)^*$ and $b(ab)^*$. We want to include all of the strings in the union $L((ab)^*) \cup L(b(ab)^*)$; this union contains all alternating strings that terminate with b. As this union can be obtained by taking each string y in $L((ab)^*)$ and including both y itself and by, it can be seen that this union is equal to $L((\epsilon + b)(ab)^*)$. In effect, placing $(\epsilon + b)$ before $(ab)^*$ allows the strings in $L((ab)^*)$ to optionally be prefixed with b. In an analogous fashion, we can extend the expression $(\epsilon + b)(ab)^*$ so as to allow its strings to optionally end with a, by concatenating $(\epsilon + a)$ to this expression. The resulting expression δ_3 has $L(\delta_3) = D$:

$$\delta_3 = (\epsilon + b)(ab)^*(\epsilon + a).$$

We can actually derive the expression δ_3 from the expression δ_2 above, as follows. First, observe that the language $L(b(ab)^*a)$ contains all non-empty alternating strings that begin with b and end with a, so the empty string is the only string in $L((ba)^*)$ that is not in $L(b(ab)^*a)$. From this observation, it follows that

$$L((ab)^* + (ba)^*) = L((ab)^* + b(ab)^*a),$$

since the empty string is in $L((ab)^*)$. We then have that

$$L(\delta_2) = L((ab)^* + b(ab)^*a + (ab)^*a + (ba)^*b).$$

From this last expression, if we replace $(ba)^*b$ with $b(ab)^*$, and reorder, we obtain

$$(ab)^* + b(ab)^* + (ab)^*a + b(ab)^*a,$$

yet another expression whose language is D. We then have the chain of equalities

$$\begin{aligned} L((ab)^* + b(ab)^* + (ab)^*a + b(ab)^*a) &= L(\epsilon(ab)^*\epsilon + b(ab)^*\epsilon + \epsilon(ab)^*a + b(ab)^*a) \\ &= L((\epsilon + b)(ab)^*\epsilon + (\epsilon + b)(ab)^*a) \\ &= L((\epsilon + b)(ab)^*(\epsilon + a)). \end{aligned}$$

This gives the claimed derivation, as the last expression appearing is exactly δ_3. In this chain, the latter two equalities can be justified algebraically by distributive laws stating that $L(\gamma(\alpha + \beta)) = L(\gamma\alpha + \gamma\beta)$ and $L((\alpha + \beta)\gamma) = L(\alpha\gamma + \beta\gamma)$, where α, β, and γ denote regular expressions. ◇

1.5.2 Regular expressions characterize the regular languages

We next establish that regular expressions give another characterization of the regular languages, in the following precise sense.

Theorem 1.5.8. *A language B is regular if and only if there exists a regular expression α such that $L(\alpha) = B$.*

That is, the languages that are representable by regular expressions are exactly the regular languages. This theorem's statement is reminiscent of the automaton-based characterizations of regularity (in Theorem 1.3.24), which each say that a language B is regular if and only if there *exists* an automaton, of some type, whose language is B. Here, instead of positing the existence of an automaton, we posit the existence of a regular expression.

This theorem can be viewed as a characterization of the regular languages *in terms of* closure properties. In essence, the theorem says that if one starts with the language $\emptyset$ and the languages $\{a\}$ for each $a \in \Sigma \cup \{\epsilon\}$, and then closes these languages under union, concatenation, and the star operator, the resulting class of languages is precisely the class of regular languages.

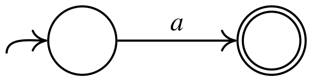


Figure 1.5.1. An ϵ-NFA whose language is $\{a\}$, for any symbol a in an alphabet or for $a = \epsilon$.

Remark 1.5.9. Recall that we proved closure of the regular languages under complementation and intersection (Theorems 1.2.1 and 1.2.2). However, these two operations are not among the operations just mentioned; they are not among the operations permitted in the evaluation of regular expressions! This observation reveals a subtlety lurking under the new characterization of regularity provided by this theorem; consider the following interesting consequences of this characterization. For each regular expression α, there exists a second regular expression α' whose language $L(\alpha')$ is the complement of the language $L(\alpha)$ of the first expression. However, in many cases, it may not be immediately obvious how to explicitly generate the expression α' from the expression α! (Indeed, the reader is invited to ponder how to do this for the example regular expressions seen so far.) Analogously, it is a consequence of the given characterization that, for any two regular expressions α and β, there exists a regular expression γ whose language $L(\gamma)$ is the intersection of $L(\alpha)$ and $L(\beta)$; but in concrete cases it may not be obvious how to generate γ from α and β.[2]

Having multiple characterizations of regularity in hand offers us the general advantage of having multiple characterizations of any property: to establish a result about regularity, we can choose which characterization to work with; a particular result may be easier to establish with one characterization than with another, and different characterizations offer different sources of illumination. See Exercise 1.9.35 for an example of a result on regular languages that can be cleanly established using Theorem 1.5.8. ◇

We prove Theorem 1.5.8 in the next two theorems, which establish the backward direction and the forward direction, respectively.

Theorem 1.5.10. *For each regular expression γ, it holds that the language $L(\gamma)$ is regular.*

Proof. We prove this result by induction on the structure of the expression γ. (Alternatively, the proof may be conceived of as by induction on the length of γ.) We consider cases, depending on the form of the expression γ.

- Suppose $\gamma = \emptyset$. The language $L(\gamma) = \emptyset$ is regular, for example, via a DFA that has no accept states.

2. The observant reader might have noticed a similar phenomenon after seeing the characterization of regularity by the NFA model. To wit: for each NFA M, there exists a second NFA M' whose language $L(M')$ is the complement of the language $L(M)$; yet, in many cases, it may not be obvious how to generate such an NFA M' from the NFA M. In general, whenever we have a language class characterized by a model, we can ask how a closure property of the class translates to an operation on realizations of the model.

- Suppose $\gamma = a$, where $a \in \Sigma \cup \{\epsilon\}$. It is straightforward to verify that the language $L(\gamma) = \{a\}$ is regular; see Figure 1.5.1 for an ϵ-NFA whose language is $\{a\}$.
- Suppose γ has the form $\alpha + \beta$. By induction, each of the languages $L(\alpha)$ and $L(\beta)$ is regular. It follows from Theorem 1.2.4 that the language $L(\gamma) = L(\alpha) \cup L(\beta)$ is regular.
- Suppose γ has the form $\alpha\beta$. By induction, each of the languages $L(\alpha)$ and $L(\beta)$ is regular. It follows from Theorem 1.4.1 that the language $L(\gamma) = L(\alpha) \cdot L(\beta)$ is regular.
- Suppose γ has the form α^*. By induction, the language $L(\alpha)$ is regular. It follows from Theorem 1.4.2 that the language $L(\gamma) = L(\alpha)^*$ is regular.
- Suppose γ has the form (α). By induction, the language $L(\alpha)$ is regular, and so the language $L(\gamma) = L(\alpha)$ is regular. □

Theorem 1.5.11. *For each regular language B, there exists a regular expression α such that $L(\alpha) = B$.*

When B is a regular language, there is an ϵ-NFA $M = (Q, \Sigma, S, T, \Delta)$ such that $L(M) = B$, by Theorem 1.3.24. To establish the theorem, we show how to pass from the automaton M to a regular expression whose language is $L(M)$.[3] We know that a string x is accepted by M if there exists a start state $s \in S$ and an accept state $t \in T$ such that $[s, x] \vdash_M^* [t, \epsilon]$. Thus, to obtain a regular expression for $L(M)$, it would be sufficient to have, for each pair (u, v) of states, a regular expression for the set of strings x such that $[u, x] \vdash_M^* [v, \epsilon]$; one could then take the sum (+) of these expressions over each pair $(s, t) \in S \times T$.

The following key definition presents a restriction of the $\vdash_M^*$ relation, and will facilitate our building the desired regular expressions by induction. Let $P \subseteq Q$, and let $u, v \in Q$ be states. When $x \in \Sigma^*$ is a string, we write

$$[u, x] \vdash_M^{P,*} [v, \epsilon]$$

when there exist configurations $[q_1, y_1], \ldots, [q_k, y_k]$ such that $[q_1, y_1] \vdash_M \cdots \vdash_M [q_k, y_k]$, $[u, x] = [q_1, y_1]$, $[q_k, y_k] = [v, \epsilon]$, $k \geq 1$, and each index j with $1 < j < k$ has $q_j \in P$. That is, the relationship $[u, x] \vdash_M^{P,*} [v, \epsilon]$ holds when the configuration $[v, \epsilon]$ can be reached from the configuration $[u, x]$ by taking successor configurations zero or more times, with the restriction that any strictly intermediate configuration must have a state from P. Note that, in this definition, the configurations $[u, x]$ and $[v, \epsilon]$ themselves need not have states from P. We can make the following observations, which hold for all configurations $[u, x]$ and $[v, \epsilon]$:

- If $[u, x] \vdash_M^0 [v, \epsilon]$ or $[u, x] \vdash_M^1 [v, \epsilon]$, then $[u, x] \vdash_M^{P,*} [v, \epsilon]$, for any subset $P \subseteq Q$.
  (That is, the relations $\vdash_M^0$ and $\vdash_M^1$ are subsets of $\vdash_M^{P,*}$, for any subset $P \subseteq Q$.)
- It holds that $[u, x] \vdash_M^* [v, \epsilon]$ if and only if $[u, x] \vdash_M^{Q,*} [v, \epsilon]$.
  (That is, the relations $\vdash_M^*$ and $\vdash_M^{Q,*}$ are equal.)

3. To establish the theorem, it would suffice to show how to pass from a DFA to a regular expression. However, as our proof technique applies quite directly to ϵ-NFA, we carry it out for this model.

The key result concerning this definition is the following.

Lemma 1.5.12. *For all $P \subseteq Q$ and $u, v \in Q$, there exists a regular expression α_{uv}^{P} where*

$$L(\alpha_{uv}^{P}) = \left\{ x \;\middle|\; [u,x] \vdash_{M}^{P,*} [v,\epsilon] \right\}.$$

Proof. We prove the result by induction on the size $|P|$ of P.

In the case that $|P| = 0$, we have that $P = \emptyset$ and thus $[u,x] \vdash_{M}^{P,*} [v,\epsilon]$ if and only if either $[u,x] \vdash_{M}^{0} [v,\epsilon]$ or $[u,x] \vdash_{M}^{1} [v,\epsilon]$. The only strings x that can satisfy this condition must have $|x| \leq 1$. So define the set of strings

$$S = \left\{ x \in \Sigma \cup \{\epsilon\} \;\middle|\; [u,x] \vdash_{M}^{0} [v,\epsilon] \text{ or } [u,x] \vdash_{M}^{1} [v,\epsilon] \right\}.$$

If S is empty, then we may take α_{uv}^{P} to be the regular expression $\emptyset$. If S is non-empty, let $a_1, \ldots, a_\ell$ be a list of its elements; then, we may take α_{uv}^{P} to be the sum $a_1 + \cdots + a_\ell$.

In the case that $|P| > 0$, fix an element $p \in P$; we claim that the regular expression

$$\alpha_{uv}^{P} = \alpha_{uv}^{P\setminus\{p\}} + \alpha_{up}^{P\setminus\{p\}}(\alpha_{pp}^{P\setminus\{p\}})^{*}\alpha_{pv}^{P\setminus\{p\}}$$

has the desired property, namely, that $L(\alpha_{uv}^{P}) = \left\{ x \;\middle|\; [u,x] \vdash_{M}^{P,*} [v,\epsilon] \right\}$. Informally, this is because when x is a string such that $[u,x] \vdash_{M}^{P,*} [v,\epsilon]$, there is a sequence of configurations witnessing this. If this sequence makes no intermediate use of the state p, then x is in the language of $\alpha_{uv}^{P\setminus\{p\}}$; otherwise, based on when the sequence visits the state p, the string x can be broken up into segments $x_0, \ldots, x_\ell$ where

- x_0 takes the ϵ-NFA from state u to state p,
- each of $x_1, \ldots, x_{\ell-1}$ takes the ϵ-NFA from state p to state p, and
- x_ℓ takes the ϵ-NFA from state p to state v,

and no visits to the state p are made other than those just mentioned explicitly; so, via the given segments, the string x is in the language of $\alpha_{up}^{P\setminus\{p\}}(\alpha_{pp}^{P\setminus\{p\}})^{*}\alpha_{pv}^{P\setminus\{p\}}$. Moreover, the reasoning reverses: when x is a string in the language of $\alpha_{uv}^{P\setminus\{p\}}$ or of $\alpha_{up}^{P\setminus\{p\}}(\alpha_{pp}^{P\setminus\{p\}})^{*}\alpha_{pv}^{P\setminus\{p\}}$, we have the relationship $[u,x] \vdash_{M}^{P,*} [v,\epsilon]$.

We verify the claim formally as follows; let x be a string.

- Suppose that $x \in L(\alpha_{uv}^{P})$; then $x \in L(\alpha_{uv}^{P\setminus\{p\}})$ or $x \in L(\alpha_{up}^{P\setminus\{p\}}(\alpha_{pp}^{P\setminus\{p\}})^{*}\alpha_{pv}^{P\setminus\{p\}})$.

  In the former case, $[u,x] \vdash_{M}^{P\setminus\{p\},*} [v,\epsilon]$, implying that $[u,x] \vdash_{M}^{P,*} [v,\epsilon]$.

  In the latter case, there exist strings $x_0, \ldots, x_\ell$ (with $\ell \geq 1$) such that $x = x_0 \ldots x_\ell$, $x_0 \in L(\alpha_{up}^{P\setminus\{p\}})$, $x_1, \ldots, x_{\ell-1} \in L((\alpha_{pp}^{P\setminus\{p\}})^{*})$, $x_\ell \in L(\alpha_{pv}^{P\setminus\{p\}})$. We thus have that

  - $[u,x_0] \vdash_{M}^{P\setminus\{p\},*} [p,\epsilon]$,
  - $[p,x_i] \vdash_{M}^{P\setminus\{p\},*} [p,\epsilon]$ for each i with $1 \leq i < \ell$, and
  - $[p,x_\ell] \vdash_{M}^{P\setminus\{p\},*} [v,\epsilon]$.

  It follows, by the same reasoning that justified Proposition 1.3.19, that

$$[u, x_0 \ldots x_\ell] \vdash_{M}^{P\setminus\{p\},*} [p, x_1 \ldots x_\ell] \vdash_{M}^{P\setminus\{p\},*} \cdots \vdash_{M}^{P\setminus\{p\},*} [p, x_\ell] \vdash_{M}^{P\setminus\{p\},*} [v,\epsilon],$$

and we thus obtain that $[u, x] = [u, x_0 \ldots x_\ell] \vdash_M^{P,*} [v, \epsilon]$.

- Suppose that $[u, x] \vdash_M^{P,*} [v, \epsilon]$ holds. If $[u, x] \vdash_M^{P \setminus \{p\},*} [v, \epsilon]$ holds, then $x \in L(\alpha_{uv}^{P \setminus \{p\}})$ holds by induction, and hence $x \in L(\alpha_{uv}^{P})$. Otherwise, there exist configurations $[p_1, y_1], \ldots, [p_k, y_k]$ of M, with $k \geq 1$, such that $p_1, \ldots, p_k \in P$ and

$$[u, x] \vdash_M [p_1, y_1] \vdash_M [p_2, y_2] \vdash_M \cdots \vdash_M [p_k, y_k] \vdash_M [v, \epsilon],$$

where the state p appears among the states $p_1, \ldots, p_k$. Let $[p, z_1]$, $[p, z_2]$, $\ldots$, $[p, z_\ell]$ be a list, in order, of the configurations among $[p_1, y_1], \ldots, [p_k, y_k]$ whose state is equal to p. We have that

  - $[u, x] \vdash_M^{P \setminus \{p\},*} [p, z_1]$,
  - $[p, z_i] \vdash_M^{P \setminus \{p\},*} [p, z_{i+1}]$ for each i with $1 \leq i < \ell$, and
  - $[p, z_\ell] \vdash_M^{P \setminus \{p\},*} [v, \epsilon]$.

From these relationships, it can be seen that the string z_1 is obtainable from x by removing zero or more symbols from the front of x; likewise, the string z_{i+1} is obtainable from z_i by removing zero or more symbols from the front of z_i (for each i with $1 \leq i < \ell$). Consequently, there exist strings $x_0, \ldots, x_\ell$ such that $x = x_0 z_1$; $z_i = x_i z_{i+1}$ (for each i with $1 \leq i < \ell$); and, $z_\ell = x_\ell$. Observe that $x = x_0 \ldots x_\ell$. By the same reasoning that justified Proposition 1.3.19, we obtain that

  - $[u, x_0] \vdash_M^{P \setminus \{p\},*} [p, \epsilon]$,
  - $[p, x_i] \vdash_M^{P \setminus \{p\},*} [p, \epsilon]$ for each i with $1 \leq i < \ell$, and
  - $[p, x_\ell] \vdash_M^{P \setminus \{p\},*} [v, \epsilon]$.

It follows by induction that $x_0 \in L(\alpha_{up}^{P \setminus \{p\}})$; $x_1, \ldots, x_{\ell-1} \in L(\alpha_{pp}^{P \setminus \{p\}})$; and, $x_\ell \in L(\alpha_{pv}^{P \setminus \{p\}})$. We derive that $x = x_0 \ldots x_\ell \in L(\alpha_{up}^{P \setminus \{p\}}(\alpha_{pp}^{P \setminus \{p\}})^* \alpha_{pv}^{P \setminus \{p\}})$ and hence that $x \in L(\alpha_{uv}^{P})$, as desired. □

Proof of Theorem 1.5.11. From Lemma 1.5.12, we obtain that, for any string $x \in \Sigma^*$, it holds that $x \in L(\alpha_{uv}^{Q})$ if and only if $[u, x] \vdash_M^{Q,*} [v, \epsilon]$. To conclude the proof, we need to give a regular expression α where $L(\alpha) = L(M)$. We have that $x \in L(M)$ if and only if there exist states $s \in S$ and $t \in T$ such that $[s, x] \vdash_M^{*} [t, \epsilon]$, or equivalently, such that $[s, x] \vdash_M^{Q,*} [t, \epsilon]$. Hence, we may define α as the sum (+) of α_{st}^{Q} over all $s \in S$ and $t \in T$. □

1.6 Proving non-regularity

The characterizations of the regular languages seen so far naturally lend themselves to establishing regularity: we can establish that a language B is regular by presenting a DFA, NFA, ϵ-NFA, or regular expression whose language is equal to B. Correspondingly, we have seen numerous positive examples of languages that are regular. This section presents tools for proving that languages are *not* regular.

A motivating example

Before presenting the general theory, we consider the particular language

$$E = \{a^n b^n \mid n \geq 0\},$$

which is perhaps the most classic example of a non-regular language. This language will serve as a running example throughout the section.

Example 1.6.1. Let us try to first gain a heuristic understanding of why the language E ought to be non-regular. As our current situation requests us to show *limits* on the scope of regular languages, it behooves us to revert to working with the simplest, most unadulterated automaton model that characterizes regularity: the DFA. So, let us try to understand why there cannot be a DFA whose language is E. A DFA can only scan a string from left to right, in one shot. Thus, intuitively speaking, a DFA for E would need to count the number n of a's that it sees prior to seeing any b, in a first phase; and then, in a second phase, make sure that the number of b's that follow is exactly equal to n. However, since a DFA by definition can only have a finite number of states, it cannot truly keep count of the number of a's seen so far, in the first phase. Roughly speaking, if one keeps on feeding a's to the DFA and monitors the states that the DFA goes through, the DFA will eventually be seen to confuse two different quantities of a's, lumping them together onto the same state.

Let us be more formal, and also set back our sights by trying to establish that there is no DFA M with 10 or fewer states whose language is E—a goal that is seemingly more modest than proving that there is no DFA whatsoever for E. Consider the 11 strings

$$a^1 = a,\ a^2 = aa,\ \ldots,\ a^{11} = aaaaaaaaaaa.$$

Assume that M is a DFA with 10 or fewer states. Then, there must exist two distinct strings, among the 11 presented, that cause the DFA to reach the same state. Precisely, there exist distinct values $i, j \in \{1, \ldots, 11\}$ and there exists a state q such that $[s, a^i] \vdash_M^* [q, \epsilon]$ and $[s, a^j] \vdash_M^* [q, \epsilon]$. Once this shared state q is reached, the DFA cannot and does not distinguish between the two strings a^i and a^j. Let us take the string b^i, which ought to cause the DFA to accept when it follows a^i. Look at the state q' that the DFA reaches after processing b^i from state q, that is, the state q' such that $[q, b^i] \vdash_M^* [q', \epsilon]$. If this state q' is not an accept state, then the DFA does not accept $a^i b^i$, which is in E; hence, the DFA's language is not E. On the other hand, if this state q' is an accept state, then the DFA does accept $a^j b^i$, which is not in E; hence, the DFA's language is not E. Either way, we can conclude that the language of the DFA M is not E.

In fact, the argument just presented generalizes perfectly; it is readily seen that, for *any* number $k \geq 1$, an analogous argument establishes that there is no DFA with k states whose language is E. From this, one may conclude that E is not regular. ◇

We next proceed to give a general framework for establishing non-regularity of languages. However, one can view this general theory as being obtained by simply abstracting out elements that are present in the argument of Example 1.6.1!

Theory

We here present notions and results which culminate in a general theorem allowing one to show non-regularity of languages.

Definition 1.6.2. With respect to a language B over an alphabet Σ, a string $w \in \Sigma^*$ is a **separator** for a pair of strings $x, y \in \Sigma^*$ if exactly one of the two strings xw, yw is in B. $\diamond$

Let us highlight that, in this definition, there is no requirement that the strings x, y be included in—or excluded from—the language B.

Example 1.6.3. Let i and j be distinct natural numbers; consider the pair of strings a^i, a^j. With respect to the language E:

- The string b^i is a separator for the pair a^i, a^j: $a^i b^i$ is in E, but $a^j b^i$ is not in E.
- The string b^j is also a separator for the pair a^i, a^j: $a^i b^j$ is not in E, but $a^j b^j$ is in E.
- When k is a natural number with $k \neq i$ and $k \neq j$, the string b^k is not a separator for the pair a^i, a^j: each of the strings $a^i b^k$, $a^j b^k$ is not in E. $\diamond$

We next argue that two strings having a separator, with respect to the language of a DFA, must be sent to different states by the DFA. (In Example 1.6.1, we used essentially this result in contrapositive form: we derived that the DFA did not decide the desired language by showing that two strings having a separator were sent to the same state.)

Proposition 1.6.4. *Let $M = (Q, \Sigma, s, T, \delta)$ be a DFA, and let $y, y' \in \Sigma^*$ be any strings. Denote by $p, p' \in Q$ the unique states such that $[s, y] \vdash_M^* [p, \epsilon]$ and $[s, y'] \vdash_M^* [p', \epsilon]$ hold. If there exists a separator for y and y' with respect to $L(M)$, then $p \neq p'$.*

Figure 1.6.1 depicts the setup of this proposition's statement.

Proof. We show the contrapositive: we assume that $p = p'$, and prove that there exists no separator for y and y'. Let $w \in \Sigma^*$ be any string. By assumption, we have $[s, y] \vdash_M^* [p, \epsilon]$ and $[s, y'] \vdash_M^* [p, \epsilon]$. We obtain $[s, yw] \vdash_M^* [p, w]$ and that $[s, y'w] \vdash_M^* [p, w]$, via Proposition 1.3.19. Let $q \in Q$ be the unique state such that $[p, w] \vdash_M^* [q, \epsilon]$. Then $[s, yw] \vdash_M^* [q, \epsilon]$ and $[s, y'w] \vdash_M^* [q, \epsilon]$. If $q \in T$, then yw and $y'w$ are both in $L(M)$; if $q \notin T$, then yw and $y'w$ are both not in $L(M)$. Hence, w is not a separator for y and y'. $\square$

Definition 1.6.5. Let us say that a set S of strings is **pairwise separable** with respect to a language B if each pair of distinct strings $x, y \in S$ have a separator with respect to B. $\diamond$

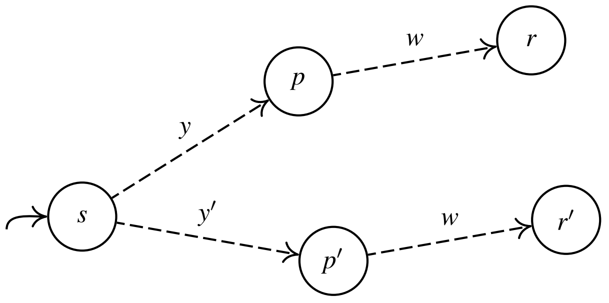


Figure 1.6.1. The setup of Proposition 1.6.4. States from a DFA M are shown. The states p and p' are the states reached by the DFA after reading the strings y and y' from the start state s, respectively. A separator for y and y', with respect to $L(M)$, is a string w such that exactly one of the two strings $yw, y'w$ is in $L(M)$; this is equivalent to the condition that exactly one of the two states r, r' is accepting, where r and r' are the states that the DFA reaches from p and p', respectively, after reading the string w. When such a separator exists, the proposition holds that the states p and p' must be distinct from each other. Note that, in this diagram, we do not indicate which of the states are accepting. Indeed, the proposition's statement and proof are agnostic about whether or not each of the states s, p, and p' is accepting.

Example 1.6.6. We consider separability with respect to the language E. The set of strings $\{a^1, a^2, \ldots, a^{11}\}$, considered in Example 1.6.1, is pairwise separable; any two distinct strings in this set have a separator, by the discussion of Example 1.6.3. Indeed, by this discussion, the infinite set of strings $\left\{a^i \mid i \geq 1\right\}$ is pairwise separable. ◇

Given a language, the following theorem allows us to establish a lower bound on the number of states of any DFA deciding the language.

Theorem 1.6.7. *Let B be a language over alphabet Σ, and let $k \geq 2$. Suppose that there exists a finite set $Y \subseteq \Sigma^*$ of size k that is pairwise separable with respect to B. Then any DFA M for which $L(M) = B$ has k or more states.*

Proof. Assume that $M = (Q, \Sigma, s, T, \delta)$ is a DFA with $L(M) = B$. Let $y_1, \ldots, y_k$ denote the strings in Y. Let $q_1, \ldots, q_k \in Q$ be the states such that $[s, y_i] \vdash_M^* [q_i, \epsilon]$, for each $i = 1, \ldots, k$. Suppose that $i, j \in \{1, \ldots, k\}$ are distinct indices; then the strings y_i and y_j have a separator w by assumption, and so by Proposition 1.6.4, it follows that $q_i \neq q_j$. Consequently, the states $q_1, \ldots, q_k$ are pairwise distinct, and $|Q| \geq k$. □

From the previous theorem, we derive a sufficient condition for showing non-regularity of a language.

Theorem 1.6.8. *Let B be a language. Suppose that there exists an infinite set Z of strings that is pairwise separable with respect to B. Then the language B is not regular.*

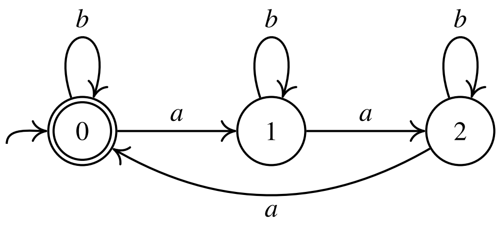


Figure 1.6.2. A DFA deciding the language A of Example 1.6.11. The language A contains a string if and only if the number of a's in the string is a multiple of 3. By swapping the roles of the symbols a and b, one obtains a DFA deciding the language B of Example 1.6.11; the language B contains a string if and only if the number of b's in the string is a multiple of 3.

Proof. We prove this by contradiction. Suppose that there exists a DFA M with $L(M) = B$. Let n denote the number of states that M has. Since the set Z is infinite, it has a subset Y of size $n + 1$. By Theorem 1.6.7, any DFA whose language is B must have $n + 1$ or more states, and we have a contradiction to $L(M) = B$. □

Applications

We next give two examples that illustrate how Theorem 1.6.8 can be used to prove the non-regularity of languages.

Example 1.6.9. Although we already argued that the language E is not regular (in Example 1.6.1), let us explain how to derive this from Theorem 1.6.8. As discussed in Example 1.6.6, the infinite set of strings $\left\{a^i \mid i \geq 1\right\}$ is pairwise separable with respect to E. Hence, by Theorem 1.6.8, we obtain that the language E is not regular. ◇

Example 1.6.10. Let rev(x) denote the reversal of a string x. We can define rev(x) inductively: $\text{rev}(\epsilon) = \epsilon$ and $\text{rev}(ya) = a \cdot \text{rev}(y)$, for all $y \in \Sigma^*$ and $a \in \Sigma$. A **palindrome** is a string x such that $x = \text{rev}(x)$, that is, a string that reads identically forwards and backwards.

Consider the language P containing all palindromes over $\{a, b\}$. We prove that this language is not regular. Set $x_i = a^i b$ for all $i \geq 1$. We show that the set of strings $\{x_1, x_2, \ldots\}$ is pairwise separable, with respect to P; this suffices by Theorem 1.6.8.

We argue this as follows. Let $i, j \geq 1$ be distinct indices. We want to show that the pair of strings $x_i = a^i b$ and $x_j = a^j b$ has a separator w. Pick $w = a^i$. We have $x_i w = a^i b a^i \in P$. And, we have $x_j w = a^j b a^i \notin P$: due to i and j being distinct, the two strings $a^j b a^i$ and $\text{rev}(a^j b a^i) = a^i b a^j$ are not equal. ◇

We next turn to deploy our development in a different way. In the following example (Example 1.6.11), we show that any DFA for a particular *regular* language must have a certain minimum number of states. We exhibit further results of this form in the subsequent example (Example 1.6.12).

Example 1.6.11. Set $\Sigma = \{a, b\}$, and let

$$C = \left\{ y \in \Sigma^* \;\middle|\; \#_a(y) \text{ and } \#_b(y) \text{ are both multiples of } 3 \right\}.$$

Recall that when d is a symbol and y is a string, $\#_d(y)$ denotes the number of occurrences of d in y. We can view C as the intersection of the following two languages:

$$A = \{y \in \Sigma^* \mid \#_a(y) \text{ is a multiple of } 3\}, \quad B = \{y \in \Sigma^* \mid \#_b(y) \text{ is a multiple of } 3\}.$$

Each of the languages A and B is regular, and has a DFA with 3 states; see Figure 1.6.2. The proof of Theorem 1.2.2 implies that the language C has a DFA with $3 \cdot 3 = 9$ states.

We prove that *any* DFA whose language is C must have 9 or more states, by using Theorem 1.6.7. Consider the set of strings $\left\{ a^i b^j \;\middle|\; i, j \in \{1, 2, 3\} \right\}$. We claim that this set is pairwise separable, which suffices, as it contains 9 strings. Let $x = a^i b^j$ and $x' = a^{i'} b^{j'}$ be distinct strings from this set; then either $i \neq i'$ or $j \neq j'$.

- First, consider the case that $i \neq i'$. It can be seen that $|i' - i|$ is equal to 1 or 2, so $i' - i$ is not a multiple of 3. We claim that the string $w = a^{3-i} b^{3-j}$ is a separator for x and x'. We have $xw = a^i b^j a^{3-i} b^{3-j}$, so $\#_a(xw) = \#_b(xw) = 3$ and $xw \in C$. On the other hand, we have $x'w = a^{i'} b^{j'} a^{3-i} b^{3-j}$, so $\#_a(x'w) = i' + 3 - i$. Since $i' - i$ is not a multiple of 3, neither is $i' + 3 - i$, so $x'w \notin C$.
- Next, consider the case that $j \neq j'$. The handling of this case is similar to that of the previous case. We argue that the same string $w = a^{3-i} b^{3-j}$ is a separator for x and x'. We have $xw = a^i b^j a^{3-i} b^{3-j} \in C$, as before. But we now have $\#_b(x'w) = j' + 3 - j$; since $j \neq j'$, we have that $j' - j$ is not a multiple of 3 and hence that $j' + 3 - j$ is not a multiple of 3, implying that $x'w \notin C$. ◇

Recall that, from two given DFA, the *product construction* (of Theorem 1.2.2) allowed us to define a DFA whose language was the intersection of the two given DFA's languages. According to this construction, the new DFA's state set is the product of the state sets of the original two DFA; hence, the new state set's size is the product of the sizes of the given state sets. Example 1.6.11 offers a perspective on the product construction; this example reveals that, in one particular case, the increase in the number of states suggested by this construction is in fact inherent. (Exercise 1.9.32 asks for a general proof that the product construction is optimal in this sense.)

The *subset construction* of a DFA from an ϵ-NFA (given by Theorem 1.3.20) involved an *exponential* increase in the number of states: the state set of the constructed DFA was the *power set* of the state set of the given ϵ-NFA. This observation naturally poses a question: is this exponential increase inherent, or is it merely an artifact of the particular proof method used? In the next example, we show that the exponentiality of the increase is inherent and necessary. That is, in the context of automata theory, nondeterministic computation is exponentially more economical than deterministic computation—when economy is measured according to how many states an automaton has. This result foreshadows a conundrum that will play primary protagonist in our study of complexity theory: there, the

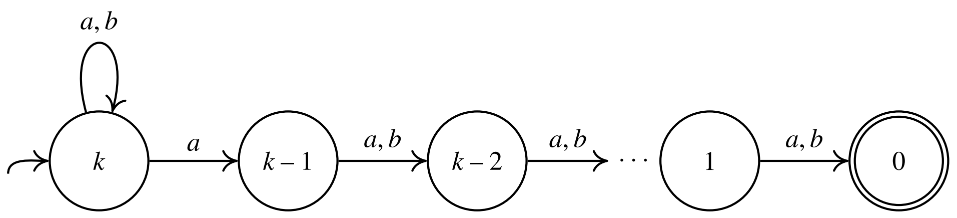


Figure 1.6.3. An NFA with $k + 1$ states whose language is B_k. The language B_k is defined to contain each string x over the alphabet $\Sigma = \{a, b\}$ such that the kth symbol from the right in x exists and is equal to a. The diagram can also be viewed as that of an ϵ-NFA having B_k as its language.

central *P versus NP* question amounts to asking whether or not nondeterministic computation is exponentially more economical than deterministic computation, when economy is measured according to how much *time* a computation takes. In particular, this question asks whether a natural exponential increase, which arises from simulating nondeterministic computation with deterministic computation, is inherent.

Example 1.6.12. Let $\Sigma = \{a, b\}$, and for each $k \geq 1$, let B_k be the language that contains each string x such that $|x| \geq k$ and a is the kth symbol from the right in x. Example 1.3.6 gave a 4-state NFA whose language is B_3. By generalizing the idea in that example, it can be seen that for each $k \geq 1$, there exists a $(k + 1)$-state NFA whose language is B_k. (By Proposition 1.3.18, there also exists a $(k + 1)$-state ϵ-NFA whose language is B_k.) Figure 1.6.3 shows such an NFA.

We prove that, for each $k \geq 1$, *any* DFA whose language is B_k must have at least 2^k states. Let Y_k be the set containing each string over Σ having length k. By appeal to Theorem 1.6.7, it suffices to argue that Y_k is pairwise separable with respect to B_k. Let $y = y_1 \ldots y_k$ and $z = z_1 \ldots z_k$ be distinct strings in Y_k. There exists an index $i \in \{1, \ldots, k\}$ such that $y_i \neq z_i$; it then holds that for the strings yb^{i-1}, zb^{i-1}, the kth symbols from the right are y_i and z_i, respectively, so exactly one of these two strings is in B_k. We have thus shown that y and z are separable, with respect to B_k. ◇

1.7 Myhill-Nerode theory

The previous section presented a *sufficient* condition for non-regularity of a language—namely, the existence of an infinite set of pairwise separable strings—and applied this condition to show the non-regularity of some particular languages. A natural question that one can ask about this condition is whether or not it is also *necessary* for non-regularity; this amounts to asking whether or not the presented proof method is *complete*, that is, whether or not it can always succeed when confronted with a non-regular language. In this section, we exhibit yet another characterization of regularity, in terms of an equivalence relation, that permits us to answer this question in the affirmative. Another fruit of our development is to give, for each regular language, a form of canonical minimal DFA, which will be used to prove the correctness of a DFA minimization algorithm in the next section. The development here originates from late 1950s work of John Myhill and Anil Nerode.

We will use the notion of *equivalence relation* and various associated concepts; we briefly review and present these now. An **equivalence relation** $\approx$ on a set U is a binary relation satisfying the following three properties.

- **Reflexivity:** for all $u \in U$, it holds that $u \approx u$.
- **Symmetry:** for all $u, v \in U$, it holds that $u \approx v$ implies $v \approx u$.
- **Transitivity:** for all $u, v, w \in U$, if $u \approx v$ and $v \approx w$, then $u \approx w$.

For each $u \in U$, define $[u]$ as $\{v \in U \mid u \approx v\}$. An **equivalence class of** $\approx$ is a set of the form $[u]$. It is known that two equivalence classes are either equal or disjoint; that is, for all $u, v \in U$, either $[u] = [v]$ or $[u] \cap [v] = \emptyset$. We say that an equivalence relation has **infinite index** if it has infinitely many equivalence classes; and **finite index** if it has finitely many equivalence classes. When an equivalence relation has finite index, its **index** is defined to be its number of equivalence classes. An equivalence relation $\approx$ on a set U **refines** a subset V of U if, for all $u, u' \in U$, it holds that $u \approx u'$ implies $u \in V \Leftrightarrow u' \in V$; equivalently, $\approx$ refines V when, for each $u \in U$, either $[u] \subseteq V$ or $[u] \cap V = \emptyset$.

Characterizing regularity via an equivalence relation

Let $B \subseteq \Sigma^*$ be a language. Define the binary relation $\sim^B$ on Σ^* as follows:

$$x \sim^B y \quad \text{if and only if} \quad \text{for all } w \in \Sigma^*, \text{ it holds that } xw \in B \Leftrightarrow yw \in B.$$

Note that the latter condition is equivalent to saying that there is *no* separator for x and y with respect to B. The relation $\sim^B$ can thus be thought of as the binary relation of *non-separability*, with respect to B. It is straightforwardly verified that $\sim^B$ is an equivalence relation (we leave this to the reader). For each $x \in \Sigma^*$, we use $[x]^B$ to denote the equivalence class of $\sim^B$ containing x, namely, the set $\{y \in \Sigma^* \mid x \sim^B y\}$. Observe that the equivalence relation $\sim^B$ refines the set B: when $x \sim^B y$, it holds that $x \in B \Leftrightarrow y \in B$, by taking $w = \epsilon$ in the definition of $x \sim^B y$.

We may observe the following proposition, which connects the index of $\sim^B$ to the notion of pairwise separability.

Proposition 1.7.1. *Let B be a language. The equivalence relation $\sim^B$ has infinite index if and only if there exists an infinite set of strings that is pairwise separable with respect to B.*

Proof. Suppose that $\sim^B$ has infinite index. Define U to be a set that contains one string from each equivalence class of $\sim^B$. Then for any two distinct strings $x, y \in U$, it does not hold that $x \sim^B y$, and thus there exists a separator for x and y with respect to B.

For the other direction, suppose that U is an infinite set of strings that is pairwise separable. Then, for any two distinct strings $x, y \in U$, it does not hold that $x \sim^B y$, and so $[x]^B \neq [y]^B$. Therefore, no two distinct strings in U fall in the same equivalence class of $\sim^B$, and thus $\sim^B$ has infinitely many equivalence classes. □

We next show that if the condition of Proposition 1.7.1 does not hold, then the language B is regular. This will allow us to characterize the notion of regularity in terms of the equivalence relation $\sim^B$.

Theorem 1.7.2. *Let B be a language. If the equivalence relation $\sim^B$ has finite index, then B is regular; in particular, there exists a DFA M^- such that $L(M^-) = B$ and whose number of states is equal to the index of $\sim^B$.*

Let B be a language over alphabet Σ such that $\sim^B$ has finite index; from B, we define a DFA $M^- = (Q^-, \Sigma, s^-, T^-, \delta^-)$ whose parts are given as follows:

$$\begin{aligned} Q^- &= \left\{[x]^B \;\middle|\; x \in \Sigma^*\right\}, \\ s^- &= [\epsilon]^B, \\ T^- &= \left\{[x]^B \;\middle|\; x \in B\right\}, \\ \delta^-([x]^B, a) &= [xa]^B. \end{aligned}$$

We have to show that δ^- is well-defined, that is, that its definition depends only on the set $[x]^B$, and not the particular representative chosen.[4] To this end, assume that $[x]^B = [x']^B$, and let $a \in \Sigma$ be arbitrary; then, for each $w \in \Sigma^*$, it holds that $xw \in B \Leftrightarrow x'w \in B$. In particular, for each $v \in \Sigma^*$, it holds that $xav \in B \Leftrightarrow x'av \in B$. It follows that $xa \sim^B x'a$ and that $[xa]^B = [x'a]^B$, as desired.

Example 1.7.3. Let B be the language $\{x \mid \#_b(x) \geq 2\}$ over $\Sigma = \{a, b\}$. (This language was previously seen, in Example 1.1.2.) Relative to this language, let us analyze the structure of the DFA M^- just given.

First, we consider the equivalence class $[\epsilon]^B$. Set $B_0 = \{x \mid \#_b(x) = 0\}$; we observe that

$$B_0 \subseteq [\epsilon]^B,$$

as when x is any string in B_0, it holds that $x \sim^B \epsilon$: for all $w \in \Sigma^*$,

$$xw \in B \quad \Leftrightarrow \quad \#_b(w) \geq 2 \quad \Leftrightarrow \quad \epsilon w \in B.$$

Set $B_1 = \{x \mid \#_b(x) = 1\}$; we observe that

$$B_1 \subseteq [b]^B,$$

4. Let us elaborate. The function δ^- needs to be defined on each pair $(q, a) \in Q^- \times \Sigma$; each element $q \in Q^-$ is an equivalence class of $\sim^B$. However, the given definition of δ^- on such a pair (q, a) is in terms of a representative element x of the equivalence class q. To ensure that this definition is proper, we thus need to verify that when x and x' are both elements of the same equivalence class q, that is, when $[x]^B = [x']^B$, the function definition yields the same result on them, that is, $[xa]^B = [x'a]^B$ holds.

as when x is any string in B_1, it holds that $x \sim^B b$: for all $w \in \Sigma^*$,

$$xw \in B \quad \Leftrightarrow \quad \#_b(w) \geq 1 \quad \Leftrightarrow \quad bw \in B.$$

Observe also that

$$B \subseteq [bb]^B,$$

as when x is any string in B, it holds that $x \sim^B bb$: for all $w \in \Sigma^*$, we have the inclusions $xw \in B$ and that $bbw \in B$.

It is straightforwardly verified that the set $\{\epsilon, b, bb\}$ of strings is pairwise separable with respect to B, implying that the equivalence classes $[\epsilon]^B$, $[b]^B$, and $[bb]^B$ are pairwise not equal. As each string in Σ^* falls into either B_0, B_1, or B, we may conclude that

$$B_0 = [\epsilon]^B, \quad B_1 = [b]^B, \quad \text{and} \quad B = [bb]^B.$$

The DFA M^- can be seen to be the bottom DFA shown in Figure 1.7.1 (on page 53). ◇

Proof of Theorem 1.7.2. Since the states in Q^- are precisely the equivalence classes of $\sim^B$, the number of states in Q^- is the index of $\sim^B$. Hence, to conclude the proof of the theorem, we need only verify that $L(M^-) = B$. Let $x = x_1 \ldots x_k \in \Sigma^*$ be an arbitrary string of length k. Then, it holds that

$$[[\epsilon]^B, x_1 \ldots x_k] \vdash_{M^-} [[x_1]^B, x_2 \ldots x_k] \vdash_{M^-} [[x_1 x_2]^B, x_3 \ldots x_k] \vdash_{M^-} \cdots \vdash_{M^-} [[x_1 \ldots x_k]^B, \epsilon].$$

If $x \in B$, then $[x]^B \in T^-$, and M^- accepts x. On the other hand, if $x \notin B$, then, as $\sim^B$ refines B, we have $[x]^B \cap B = \emptyset$ and hence $[x]^B \notin T^-$; so, M^- rejects x. □

Collecting together the above results, we obtain yet another characterization of the regular languages.

Theorem 1.7.4. *A language B is regular if and only if the equivalence relation $\sim^B$ has finite index.*

Proof. If the equivalence relation $\sim^B$ has infinite index, then it follows from Proposition 1.7.1 and Theorem 1.6.8 that B is not regular. If the equivalence relation $\sim^B$ has finite index, then it follows from Theorem 1.7.2 that B is regular. □

At this point, we can conclude that the sufficient condition for non-regularity presented in the previous section is also necessary, that is, we can establish the converse of Theorem 1.6.8. This result follows directly from Theorem 1.7.4 and Proposition 1.7.1.

Corollary 1.7.5. *Suppose that B is a non-regular language; then, there exists an infinite set of strings that is pairwise separable with respect to B.*

Minimality and canonicity of automata

In the rest of this section, we perform a more detailed study of the DFA M^-, and show that it is a canonical minimal DFA, in a sense made precise. We require the following notions. Let us say that a state q of a DFA M is **reachable** if there exists a string x such that $[s, x] \vdash_M^* [q, \epsilon]$, where s denotes the start state of M. Clearly, states of a DFA that are not reachable can be eliminated without affecting the DFA's behavior.

We next define the notion of a *homomorphism* from a DFA to another DFA. This notion allows us to structurally relate two DFA; in substance, when there exists a homomorphism from a first DFA to a second DFA, the structure of the first DFA is embodied in the structure of the second DFA.

Definition 1.7.6. When $M = (Q, \Sigma, s, T, \delta)$ and $M' = (Q', \Sigma, s', T', \delta')$ are DFA over the same alphabet, a **homomorphism** from M to M' is a map $h\colon Q \to Q'$ such that

- $h(s) = s'$;
- $q \in T \Leftrightarrow h(q) \in T'$, for all $q \in Q$; and,
- $h(\delta(q, a)) = \delta'(h(q), a)$, for all $q \in Q$ and $a \in \Sigma$. ◇

Figure 1.7.1 discusses an example of a homomorphism.

We next establish a theorem essentially showing that, for a regular language B, the structure of any DFA M for B is manifest in the structure of the DFA M^-: precisely, we show that there is a homomorphism from M to M^-.

Theorem 1.7.7. *Let B be a regular language, and let M^- be defined from B as described above; M^- is a DFA via Theorem 1.7.4. If $M = (Q, \Sigma, s, T, \delta)$ is a DFA with $L(M) = B$ and whose states are all reachable, then there exists a unique homomorphism $h\colon Q \to Q^-$ from M to M^-, and moreover, this homomorphism is surjective.*

Both here and in the sequel, we will make use of an equivalence relation $\sim_M$ on Σ^* derived from a DFA $M = (Q, \Sigma, s, T, \delta)$, defined as follows: $x \sim_M y$ if and only if there exists a state $q \in Q$ such that $[s, x] \vdash_M^* [q, \epsilon]$ and $[s, y] \vdash_M^* [q, \epsilon]$. It is straightforward (and left to the reader) to verify that this binary relation is an equivalence relation.

Proof of Theorem 1.7.7. Let $M = (Q, \Sigma, s, T, \delta)$ be a DFA satisfying the hypotheses.

We show that $\sim_M$ is a subset of $\sim^B$, as follows. Suppose that $x \sim_M y$. Then, via Proposition 1.3.19, there exists a state q such that, for each $w \in \Sigma^*$, it holds that $[s, xw] \vdash_M^* [q, w]$ and $[s, yw] \vdash_M^* [q, w]$; hence, for each $w \in \Sigma^*$, it holds that $xw \in L(M) \Leftrightarrow yw \in L(M)$. Thus $x \sim^B y$. (Consequently, each equivalence class of $\sim_M$ is contained in an equivalence class of $\sim^B$, and the index of $\sim^B$ is less than or equal to the index of $\sim_M$, which in turn is the number of states of M; recall our assumption that each state of M is reachable.)

Suppose that g is a homomorphism from M to M^-, and let $q \in Q$ be a state of M. Assume $x = x_1 \ldots x_n$ to be a string such that $[s, x] \vdash_M^* [q, \epsilon]$. We claim that $g(q) = [x]^B$. Let $q_0 = s$, and let $q_1, \ldots, q_n \in Q$ be the states such that

$$[q_0, x_1 \ldots x_n] \vdash_M [q_1, x_2 \ldots x_n] \vdash_M [q_2, x_3 \ldots x_n] \vdash_M \cdots \vdash_M [q_n, \epsilon].$$

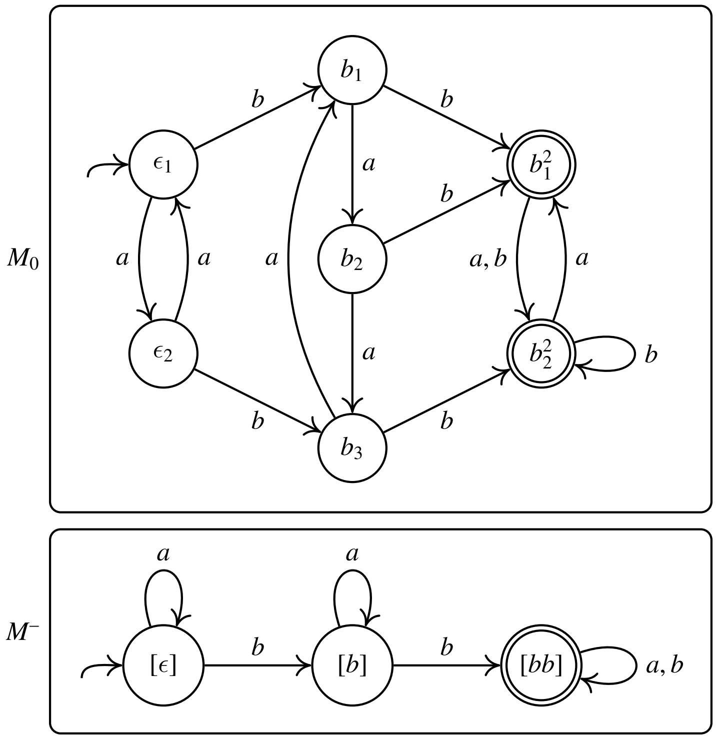


Figure 1.7.1. A pair of DFA related by homomorphism; the bottom DFA M^- is from Example 1.7.3. There is a homomorphism from the top DFA M_0 to the bottom DFA M^-, namely, the mapping h from the top DFA's state set to the bottom DFA's state set defined by $h(\epsilon_1) = h(\epsilon_2) = [\epsilon]$, $h(b_1) = h(b_2) = h(b_3) = [b]$, and $h(b_1^2) = h(b_2^2) = [bb]$. In the definition of homomorphism, the first condition holds that the start state of the first DFA is mapped to the start state of the second; here, we indeed have $h(\epsilon_1) = [\epsilon]$. The second condition holds that a state of the first DFA accepts if and only if it is mapped to an accept state; here, we have that $[bb]$ is the unique accept state of the bottom DFA, and that the states mapped to it, namely b_1^2 and b_2^2, are indeed the only accept states of the top DFA. The third condition holds that, for any state of the first DFA, transitioning on a symbol and passing through the homomorphism yields the same state as first passing through the homomorphism and then transitioning on the same symbol. As one example, consider the state ϵ_2 of the top DFA; transitioning on b leads to the state b_3, and passing through the homomorphism yields the state $[b]$; passing the state ϵ_2 through the homomorphism to obtain $[\epsilon]$ and then transitioning on b also yields the state $[b]$.

Observe that, for each i with $0 \leq i < n$, it holds that

$$g(q_{i+1}) = g(\delta(q_i, x_{i+1})) = \delta^-(g(q_i), x_{i+1}),$$

where the second equality holds by the definition of homomorphism. By the definition of homomorphism, we have $g(q_0) = [\epsilon]^B$. By repeatedly applying our observation about the value of $g(q_{i+1})$ and using the definition of δ^-, we obtain

$$g(q_1) = [x_1]^B,\ g(q_2) = [x_1x_2]^B,\ \ldots,\ g(q_n) = [x_1 \ldots x_n]^B.$$

Since $q_n = q$, we obtain $g(q) = [x]^B$, as desired.

For each $q \in Q$, define $h(q) = [x]^B$ where x is a string such that $[s, x] \vdash_M^* [q, \epsilon]$. We have that h is well-defined: for any two strings x, x' such that $[s, x] \vdash_M^* [q, \epsilon]$ and $[s, x'] \vdash_M^* [q, \epsilon]$, it holds that $x \sim_M x'$; this implies that $x \sim^B x'$ (as just shown) and that $[x]^B = [x']^B$. We have argued that any homomorphism g must be equal to h. Therefore, if h is indeed a homomorphism from M to M^-, it is the unique such homomorphism.

We verify that h is a homomorphism from M to M^- as follows.

- As $[s, x] \vdash_M^* [q, \epsilon]$ when $q = s$ and $x = \epsilon$, we have $h(s) = [\epsilon]^B$, so we obtain $h(s) = s^-$.
- Let $q \in Q$; there exists a string x such that $[s, x] \vdash_M^* [q, \epsilon]$, and $h(q) = [x]^B$.
  - When $q \in T$, the DFA M accepts x, and $x \in B$; then, $[x]^B \in T^-$.
  - When $q \notin T$, the DFA M rejects x, and $x \notin B$; since $\sim^B$ refines B, we have $[x]^B \notin T^-$.
- Let $q \in Q$ and $a \in \Sigma$. Set $x \in \Sigma^*$ so that $[s, x] \vdash_M^* [q, \epsilon]$; then $h(q) = [x]^B$ holds. We have $[s, xa] \vdash_M^* [q, a] \vdash_M [\delta(q, a), \epsilon]$, so $h(\delta(q, a)) = [xa]^B = \delta^-([x]^B, a) = \delta^-(h(q), a)$.

It holds that $h \colon Q \to Q^-$ is a surjective mapping: for any string y, let p be the state such that $[s, y] \vdash_M^* [p, \epsilon]$; then, $h(p) = [y]^B$. □

Define a DFA M to be **minimal** if there does not exist a DFA M' that has strictly fewer states than M and has $L(M') = L(M)$. That is, a DFA M is minimal if there is no strictly smaller DFA that has the same language, where we measure the size of a DFA according to the number of states.

Let B be a language, and consider a DFA M with $L(M) = B$. Suppose that the DFA M is minimal; since it is minimal, it clearly has only reachable states. Theorem 1.7.7 implies that the DFA M admits a surjective homomorphism to the DFA M^- defined from B; by the surjectivity, M has at least as many states as M^-. It follows that the DFA M^- is minimal, and so we have established the following corollary.

Corollary 1.7.8. *Let B be a regular language, and let M^- be the DFA described above. The DFA M^- is minimal.*

Remark 1.7.9. When M and M' are DFA over the same alphabet, define an **isomorphism** from M to M' to be a homomorphism from M to M' that is bijective. It is straightforwardly verified that if i is an isomorphism from M to M', then its inverse i^{-1} is an isomorphism from M' to M. Say that a DFA is **isomorphic** to another DFA when there exists an isomorphism from one to the other. Diagramatically, two DFA that are isomorphic are the same, up to relabeling the names of their states. It can be verified that relating together each pair of isomorphic DFA yields an equivalence relation (on the class of all DFA).

From Theorem 1.7.7, we learn that any minimal DFA M for a language B is isomorphic to the DFA M^-: when M is minimal, the DFA M and M^- must have the same number of states, implying that the homomorphism provided by the theorem is a bijection and hence an isomorphism. Thus, we can conceive of M^- as a canonical minimal DFA for B. ◇

1.8 DFA minimization

In this section, we present and study an algorithm that, given as input a DFA M, outputs a minimal DFA whose language is that of M.

We employ the following notation in this section. Relative to a DFA $M = (Q, \Sigma, s, T, \delta)$, when $q \in Q$ is a state and $x \in \Sigma^*$ is a string, we use $\widehat{\delta}(q, x)$ to denote the unique state such that $[q, x] \vdash_M^* [\widehat{\delta}(q, x), \epsilon]$. In words, $\widehat{\delta}(q, x)$ denotes the state that the DFA ends up in if it begins in state q and processes the string x. We will invoke the property that, for each state $q \in Q$, each symbol $a \in \Sigma$, and each string $w \in \Sigma^*$, it holds that $\widehat{\delta}(\delta(q, a), w) = \widehat{\delta}(q, aw)$. This property is straightforwardly verified, and in fact could be used to alternatively define the function $\widehat{\delta}$ by induction.

The input to the algorithm is a DFA $M = (Q, \Sigma, s, T, \delta)$. The algorithm performs three phases, in order:

- the *preliminary phase*,
- the *marking phase*, and
- the *collapsing phase*, which outputs the minimized DFA.

In the preliminary phase, the algorithm removes from M each state that is not reachable. This can be done by flagging the start state, and then iteratively flagging each state admitting a transition from a flagged state; when no more states can be flagged, the reachable states will be precisely those that are flagged, and the non-flagged states can be removed. The marking phase and collapsing phase are described and studied in what follows.

1.8.1 Marking phase

The marking phase iteratively marks elements of $Q \times Q$, that is, pairs of states. It is assumed that all pairs are unmarked prior to the commencement of this phase. The marking phase performs the following:

- Initialization: mark all pairs in $T \times (Q \setminus T)$ and in $(Q \setminus T) \times T$.
- Loop, doing the following until no more changes can be made:
  - For each unmarked pair (p, q), if there exists an element $a \in \Sigma$ such that $(\delta(p, a), \delta(q, a))$ is marked, then mark (p, q).

Intuitively, this phase marks each state pair whose states are behaviorally different, and cannot be collapsed together. The initialization marks each state pair where exactly one state is an accept state; the states of such a pair behave differently. The loop marks a state pair (p, q) as different when making a transition from these states, based on a common symbol, would lead to a pair of states already marked as being different.

Let us introduce some symmetric binary relations, each of which is a subset of $Q \times Q$, for the sake of reasoning about this algorithm. First, define

$$R_0 = (T \times (Q \setminus T)) \cup ((Q \setminus T) \times T).$$

The relation R_0 contains the pairs that are marked after the initialization step. Next, for each value $i \geq 0$, define the relation

$$R_{i+1} = R_i \cup \{(p, q) \in Q \times Q \mid \exists a \in \Sigma \text{ such that } (\delta(p, a), \delta(q, a)) \in R_i\}.$$

Clearly, we have the inclusions

$$R_0 \subseteq R_1 \subseteq R_2 \subseteq \cdots .$$

What is the meaning of the relations R_{i+1}? The relation R_{i+1} contains those pairs that are marked after the $(i + 1)$th iteration of the loop body, so long as this iteration is performed. The loop terminates as soon as no changes can be made; letting k be the lowest value such that $R_k = R_{k+1}$, the loop terminates after $k + 1$ executions of the loop body. Observe that, for this value k, it holds that $R_k = R_{k+1} = R_{k+2} = \cdots$. We nonetheless define R_{i+1} for all values $i \geq 0$, for the purpose of analyzing the algorithm.

Figure 1.8.1 discusses an example of the marking phase's behavior.

Define $R = \bigcup_{j \geq 0} R_j$. Observe that R contains a pair if and only if the pair is marked by the algorithm. We will use $\overline{R}$ to denote the complement of R with respect to $Q \times Q$; that is, we use $\overline{R}$ to denote the set $(Q \times Q) \setminus R$. We thus have that the pairs in $\overline{R}$ are precisely the pairs that are not marked by the algorithm. The following lemma provides a characterization of the set $\overline{R}$, and thus implicitly, a characterization of the set R, as well.

Lemma 1.8.1. *A pair (p, q) of states is in $\overline{R}$ if and only if for all $w \in \Sigma^*$, it holds that $\widehat{\delta}(p, w) \in T \Leftrightarrow \widehat{\delta}(q, w) \in T$. Consequently, the binary relation $\overline{R} \subseteq Q \times Q$ is an equivalence relation.*

Proof. The second statement about $\overline{R}$ being an equivalence relation is readily verified from the first statement. We prove the first statement. Let (p, q) be an arbitrary pair of states of M.

We prove the forward direction by establishing its contrapositive. Suppose that there exists $w \in \Sigma^*$ such that exactly one of $\widehat{\delta}(p, w)$, $\widehat{\delta}(q, w)$ is in T; we prove that (p, q) is not in $\overline{R}$. Let $w = w_1 \ldots w_n$. Set $p_0 = p$ and $q_0 = q$. For each index $i = 1, \ldots, n$ in sequence, define the two states $p_i = \delta(p_{i-1}, w_i)$ and $q_i = \delta(q_{i-1}, w_i)$. We have

$$[p, w_1 \ldots w_n] \vdash_M [p_1, w_2 \ldots w_n] \vdash_M [p_2, w_3 \ldots w_n] \vdash_M \cdots \vdash_M [p_n, \epsilon],$$

$$[q, w_1 \ldots w_n] \vdash_M [q_1, w_2 \ldots w_n] \vdash_M [q_2, w_3 \ldots w_n] \vdash_M \cdots \vdash_M [q_n, \epsilon].$$

	ϵ_1	ϵ_2	b_1	b_2	b_3	b_1^2	b_2^2
ϵ_1			1	1	1	0	0
ϵ_2			1	1	1	0	0
b_1	1	1				0	0
b_2	1	1				0	0
b_3	1	1				0	0
b_1^2	0	0	0	0	0		
b_2^2	0	0	0	0	0		

Figure 1.8.1. The marking phase's behavior on the DFA M_0 of Figure 1.7.1. This DFA has state set $Q = \{\epsilon_1, \epsilon_2, b_1, b_2, b_3, b_1^2, b_2^2\}$ and accept state set $T = \{b_1^2, b_2^2\}$. The relation R_0 is defined to contain all pairs that are initially marked, which are the pairs containing one accept state and one non-accept state; these pairs are indicated with a 0 in the table. The relation R_1 contains all pairs marked by the 1st iteration of the loop, as well as the pairs in R_0; in this case, R_1 newly includes each pair of the form (ϵ_i, b_j) and its transposition, as for such a pair we have $(\delta(\epsilon_i, b), \delta(b_j, b)) \in R_0$. The pairs in R_1 but not in R_0 are indicated with a 1 in the table. After the first iteration of the loop, no further pairs are marked by the marking phase, and we have $R_1 = R_2 = \cdots$. When the marking phase is concluded, the unmarked pairs always form an equivalence relation on the state set; this is established by Lemma 1.8.1. Here, this equivalence relation has the equivalence classes $\{\epsilon_1, \epsilon_2\}$, $\{b_1, b_2, b_3\}$, and $T = \{b_1^2, b_2^2\}$.

By hypothesis, exactly one of the states p_n, q_n is in T. Hence, we have $(p_n, q_n) \in R_0$. Since $(p_n, q_n) = (\delta(p_{n-1}, w_n), \delta(q_{n-1}, w_n))$, it follows from the definition of R_1 that the inclusion $(p_{n-1}, q_{n-1}) \in R_1$ holds. Repeating this reasoning, we obtain that $(p_{n-i}, q_{n-i}) \in R_i$ for each index $i = 0, \ldots, n$, and so $(p, q) = (p_0, q_0) \in R_n \subseteq R$. We have shown that (p, q) is not in $\overline{R}$.

We prove the backward direction also by establishing its contrapositive. Suppose the inclusion $(p, q) \in R$; we show that there exists $w \in \Sigma^*$ such that exactly one of the two states $\widehat{\delta}(p, w)$, $\widehat{\delta}(q, w)$ is in T. We prove by induction that for each $i \geq 0$, if $(p, q) \in R_i$, then there exists a string w with the stated property. In the case of $i = 0$, it is clear that the string $w = \epsilon$ has the stated property. For the induction, assume that the statement holds for i; we show that it holds for $i + 1$. Let $(p, q) \in R_{i+1}$. If $(p, q) \in R_i$, we are finished, by the induction hypothesis. Otherwise, by the definition of R_{i+1}, there exists $a \in \Sigma$ such that $(\delta(p, a), \delta(q, a)) \in R_i$. By the induction hypothesis, there exists a string w' such that the set T contains exactly one of the two states $\widehat{\delta}(\delta(p, a), w')$, $\widehat{\delta}(\delta(q, a), w')$. As we have the equalities $\widehat{\delta}(\delta(p, a), w') = \widehat{\delta}(p, aw')$ and $\widehat{\delta}(\delta(q, a), w') = \widehat{\delta}(q, aw')$, we can take $w = aw'$. ◻

1.8.2 Collapsing phase

The collapsing phase of the algorithm computes and outputs a minimized DFA N, described as follows. Let us use $[q]$ to denote the equivalence class of a state $q \in Q$ with respect to the equivalence relation $\overline{R}$, the set of unmarked pairs. Essentially, the DFA N is obtained from the DFA $M = (Q, \Sigma, s, T, \delta)$ by taking each equivalence class $[q]$, and collapsing all of its states into one state. Intuitively, this is justified due to all states in an equivalence class $[q]$ being behaviorally equivalent. To be formal, let us define the DFA $N = (Q_N, \Sigma, s_N, T_N, \delta_N)$ as follows:

$$\begin{aligned} Q_N &= \{[q] \mid q \in Q\}, \\ s_N &= [s], \\ T_N &= \{[t] \mid t \in T\}, \\ \delta_N([q], a) &= [\delta(q, a)]. \end{aligned}$$

We need to argue that the transition function δ_N is well-defined; in particular, we need to argue that, assuming $a \in \Sigma$, if $(p, q) \in \overline{R}$, then $(\delta(p, a), \delta(q, a)) \in \overline{R}$. We prove the contrapositive. Suppose that $(\delta(p, a), \delta(q, a)) \in R$. Then there exists an index $j \geq 0$ such that $(\delta(p, a), \delta(q, a)) \in R_j$. But then it follows by the definition of the relations R_i that $(p, q) \in R_{j+1}$, and hence $(p, q) \in R$.

In the next two theorems, the correctness of the algorithm is established. We assume, for these two theorems, that M is a DFA whose states are all reachable, and that N is the DFA defined from M as just described. These theorems show that the new DFA N has the same language as the original DFA M, and that the new DFA N is minimal.

Theorem 1.8.2. *$L(M) = L(N)$.*

Proof. Let $x = x_1 \ldots x_k$ be an arbitrary string. Let $q_1, \ldots, q_k \in Q$ be the states such that

$$[s, x_1 \ldots x_k] \vdash_M [q_1, x_2 \ldots x_k] \vdash_M [q_2, x_3 \ldots x_k] \vdash_M \cdots \vdash_M [q_k, \epsilon].$$

From the definition of δ_N, it follows that

$$[[s], x_1 \ldots x_k] \vdash_N [[q_1], x_2 \ldots x_k] \vdash_N [[q_2], x_3 \ldots x_k] \vdash_N \cdots \vdash_N [[q_k], \epsilon].$$

We show that $x \in L(M)$ if and only if $x \in L(N)$. In order to do this, we argue that for any state $q_k \in Q$, it holds that $q_k \in T$ if and only if $[q_k] \in T_N$. The forward direction follows immediately from the definition of T_N. For the backward direction, suppose that $[q_k] \in T_N$; then, there exists $t \in T$ such that $[t] = [q_k]$, that is, such that $(t, q_k) \in \overline{R}$. But by Lemma 1.8.1, we have

$$t = \widehat{\delta}(t, \epsilon) \in T \quad \Leftrightarrow \quad q_k = \widehat{\delta}(q_k, \epsilon) \in T,$$

so it follows that $q_k \in T$. □

Theorem 1.8.3. *The DFA N is minimal.*

Proof. By our assumption that all states of M are reachable, it follows that all states of N are reachable. Let $B = L(N)$; by Theorem 1.8.2, we have that $B = L(M)$. We will prove the claim that $\sim^B$ is a subset of $\sim_N$. This implies that each equivalence class of $\sim^B$ is contained in an equivalence class of $\sim_N$, which in turn implies that the index of $\sim^B$ is greater than or equal to the index of $\sim_N$. The index of $\sim^B$ is equal to $|Q^-|$, the number of states of M^-

(defined from B in the previous section). The index of $\sim_N$ is equal to $|Q_N|$, the number of states of N; this holds since all states of N are reachable. Since M^- is known to be minimal (by Corollary 1.7.8), it follows that N is minimal. (We remark that the proof of Theorem 1.7.7 yields that $\sim_N$ is a subset of $\sim^B$, here implying that $\sim_N$ and $\sim^B$ are equal.)

To establish the claim, assume that $x \sim^B y$; then, by definition we have that for each string $w \in \Sigma^*$, it holds that

$$xw \in B \quad \Leftrightarrow \quad yw \in B.$$

Since $B = L(M)$, we obtain that, for each string $w \in \Sigma^*$, it holds that

$$\widehat{\delta}_M(s, xw) \in T \quad \Leftrightarrow \quad \widehat{\delta}_M(s, yw) \in T,$$

and consequently that

$$\widehat{\delta}_M(\widehat{\delta}_M(s, x), w) \in T \quad \Leftrightarrow \quad \widehat{\delta}_M(\widehat{\delta}_M(s, y), w) \in T.$$

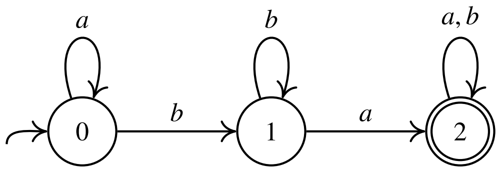
By Lemma 1.8.1, we obtain that $[\widehat{\delta}_M(s, x)] = [\widehat{\delta}_M(s, y)]$. It follows from the first paragraph of the proof of the previous theorem (Theorem 1.8.2) that

$$\delta_N([s], x) = [\widehat{\delta}_M(s, x)] \quad \text{and} \quad \delta_N([s], y) = [\widehat{\delta}_M(s, y)].$$

Therefore, we have that $\delta_N([s], x) = \delta_N([s], y)$ which, by definition of $\sim_N$, yields that the relationship $x \sim_N y$ holds. □

1.9 Exercises and notes

Exercise 1.9.1. List each string of length 3 or less that is accepted by the following DFA:



◇

Note 1.9.2: Automata as finite descriptions of languages. Observe that an automaton (of one of the defined brands) is an inherently finite object: from the supposition that each automaton has a finite state set, each part of an automaton may be presented by expending a finite amount of ink on paper. Likewise, a regular expression is by definition a string having finite length, and is also a finite object in this sense.

On the other hand, each automaton and regular expression describes a language, which is potentially an infinite object, in that there are infinitely many strings that may belong to a language, and presenting a particular language involves specifying (explicitly or implicitly) which of those strings are members. So, automata and regular expressions may be viewed as *finite* descriptions of generally *infinite* objects. ◇

Note 1.9.3. To continue the discussion of Note 1.9.2, fix an alphabet Σ. A corollary of the observation that each DFA admits a finite representation is that the number of possible DFA over Σ is countably infinite. This can be seen, in fact, by observing that each DFA (over Σ) can itself be represented as a string over an alphabet, and that the number of strings over any particular alphabet is countably infinite.

As a consequence, one obtains from size considerations that there exists a language not decided by any DFA (that is, a language that is not regular), for the number of languages over Σ is uncountably infinite. Indeed, the vast majority of languages are inaccessible in that they lack representation via DFA. The same phenomenon will persist throughout our study; for each of the studied computational models, the number of particular realizations of the model will be countably infinite, implying immediately that there are languages without representation in the model. ◇

Note 1.9.4. While the argument of Note 1.9.3 imparts that there *exist* languages that are not regular, it does not at all render uninteresting the techniques we developed for proving non-regularity of languages. The developed techniques allow us to explicitly present natural specimens of non-regular languages; more generally, they offer the possibility of analyzing whether a given language of relevance is regular or not.

As our study progresses, we will identify further classes of languages; one broad goal of the theory of computation is to understand, when confronted with a relevant language (possibly arising from the real world!), to which classes it belongs and to which classes it does not, thereby clarifying the computational resources demanded by language. ◇

Note 1.9.5: On the trivial languages and DFA. Relative to an alphabet Σ, the trivial languages $\emptyset$ and Σ^* are indeed among the most innocuous languages with which we will deal. They are the only two languages that are decidable by 1-state DFA, a fact which assures us that monikering them as *trivial* was reasonable. They fall into the families of *finite* and *cofinite* languages (respectively), which are the principals of Exercise 1.9.7. ◇

Note 1.9.6. As has already been suggested, the regular languages are the smallest class of languages and the least difficult languages that we will study computationally. That the trivial languages are clearly regular suggests that the subset and superset relations will, in general, not be of high utility for comparing the difficulty of languages. For the trivial languages bookend all other languages: over an alphabet Σ, each language is a subset of Σ^*, and a superset of the empty set $\emptyset$. ◇

Exercise 1.9.7: Regularity of finite languages. Prove that each finite language is regular.

This implies, via Theorem 1.2.1, that each *cofinite* language is also regular; a cofinite language is a language whose complement is finite. ◇

Exercise 1.9.8: On infinite state sets. It is certainly a legitimate move to define a mathematical object more general than the DFA, by lifting off the assumption that the state set be finite. Define a **deterministic automaton (DA)** in exactly the same way that a DFA is defined, but without the restriction that the state set be finite. For a DA $M = (Q, \Sigma, s, T, \delta)$, we can define the notions of *configuration*, *successor configuration*, *acceptance*, and *rejection* just as we did for DFA.

Prove that for *every* language B, there exists a DA M with $L(M) = B$. ◇

Note 1.9.9. The result of Exercise 1.9.8 highlights the cruciality of the assumption that each DFA has a *finite* state set. In the absence of this assumption, the computational model becomes trivialized in that its power becomes overwhelming: every language becomes describable. Correspondingly, the theory goes flat, not permitting any interest in or technique for showing the non-describability of languages; also, the describable languages trivially possess any closure property. ◇

Exercise 1.9.10: Symmetric differences of regular languages. The **symmetric difference** of two languages B and C is defined as the set $(B \setminus C) \cup (C \setminus B)$; it is the language containing each string that is in exactly one of B and C.

Prove that when B and C are regular languages over the same alphabet Σ, their symmetric difference is also a regular language. ◇

Note 1.9.11. When one has a language B of interest in hand, each string over the language's alphabet poses a question to a potential DFA for deciding the language B: is the string inside B? Whether or not there is a DFA deciding B is not affected by modifying the answer to this decision question for a finite number of strings: by Exercises 1.9.7 and 1.9.10, regularity of a language is preserved under taking a symmetric difference with a finite language. And, this statement holds not just for the DFA model, but for each of the principal computational models that we will consider in this book.

What we are building, then, is not so much a theory of individual decision questions, but rather, a theory of how decision questions behave in an aggregate fashion and in relation to each other. ◇

Note 1.9.12. In some of the following exercises, you are asked to present an automaton or a regular expression whose language is a given one. Also, strive for comprehensibility: to the extent possible, design and present automata and regular expressions so that their functionality is transparent and readily graspable. And, with comprehensibility in mind, strive for brevity: when presenting automata, attempt to use as few states as is feasible, and when presenting regular expressions, try to minimize the length of expressions. ◇

Exercise 1.9.13: Building DFA. Let $\Sigma = \{a, b\}$. For each of the languages over Σ that is given below, present a DFA with input alphabet Σ whose language is the given one.

1. The language containing each string with exactly 4 occurrences of b.
2. The language containing each string that ends with either aa or ab.
3. The language containing each string x such that $|x|$ is a multiple of 4.

4. The language containing each string x such that $|x|$ is a multiple of 2 or 3.
5. The language containing each string x such that $\#_a(x)$ is odd.
6. The language containing each string x such that $\#_a(x)$ is odd, equal to 2, or equal to 6.
7. The language containing each string that does not have *abb* as a substring.
8. The language containing each string x such that each occurrence of *baab* as a substring is either followed by *aa*, or is at the end of x.
9. The language containing each string x such that *ab* is a substring of x if and only if *ba* is a substring of x.
10. The language containing each string x where, for each prefix w of x, it holds that $|\#_a(w) - \#_b(w)| \leq 3$.
11. The language containing each string x where, for each prefix w of x, it holds that $|2\#_a(w) - \#_b(w)| \leq 3$.
12. The language containing each string that contains *ab* as a substring an odd number of times.
13. The language containing each string that ends with either *aaa*, *aab*, or *aba*.
14. The language containing each string that contains exactly two *b*'s or contains an odd number of *a*'s.
15. The language containing each string that contains exactly two *b*'s and contains an odd number of *a*'s. ◇

Exercise 1.9.14: Building more DFA. Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. For each of the languages over Σ that is given below, present a DFA with input alphabet Σ whose language is the given one.

1. The language B containing each string that is a number between 1 and 420, inclusive, written without any leading 0's. As examples, $0 \notin B$, $04 \notin B$, $1 \in B$, $4 \in B$, $44 \in B$, $50 \in B$, $050 \notin B$, $404 \in B$, $420 \in B$, $444 \notin B$, and $500 \notin B$.
2. The language C containing each string that is a number between 1 and 2046, inclusive, written without any leading 0's. As examples, $0 \notin C$, $042 \notin C$, $42 \in C$, $142 \in C$, $0142 \notin C$, $1024 \in C$, and $2401 \notin C$.
3. The language D containing each string that is a number between 1984 and 2001, inclusive, written without any leading 0's. ◇

Exercise 1.9.15: Building NFA. Let $\Sigma = \{a, b, c\}$. For each of the languages over Σ that is given below, present an NFA with input alphabet Σ whose language is the given one.

1. The language containing each string that contains both *ca* and *bb* as substrings.
2. The language containing each string that contains at least one of *aab* or *aac* as a substring.
3. The language containing each string x such that the second symbol from the right in x exists, and is equal to the second symbol from the left in x.

4. The language containing each string x such that if the 5th symbol from the right exists in x, it is equal to c.
5. The language containing each string having 3 consecutive symbols that are equal.
6. The language containing each string x such that there exist natural numbers $k, \ell \geq 0$ where $|x| = 5k + 7\ell$.
7. The language containing each string where each occurrence of a is followed immediately by an occurrence of b. ◇

Exercise 1.9.16: Building more NFA. Let $\Sigma = \{a, b\}$. For each of the languages over Σ that is given below, present an NFA with input alphabet Σ whose language is the given one.

1. The language containing each string x such that $|x| \geq 1$ and the first symbol of x is equal to the last symbol of x.
2. The language B containing each string x such that $|x| \geq 4$ and the first two symbols of x are equal to the last two symbols of x. As examples, $aaa \notin B$, $abab \in B$, and $abba \notin B$.
3. The language C containing each string x such that $|x| \geq 2$ and the first two symbols of x are equal to the last two symbols of x. As examples, $aaa \in C$, $aba \notin C$, $abab \in C$, and $abba \notin C$. ◇

Exercise 1.9.17. List all strings of length 3 and of length 4 that do not belong to the language $L(a^*b^*a^*)$. ◇

Exercise 1.9.18: Describing languages of regular expressions. For each of the following regular expressions, give a natural language description of the language represented, and list all of the strings of length 6 or less that belong to the language represented. If there are more than 15 such strings, then you may list the first 15 strings in a length-ascending order.

1. $(ab)^* + (ba)^*$
2. $(ab)^*c + (bc)^*$
3. $(aab)^* + (bba)^*$
4. $(a + bb)^* + (b + aa)^*$
5. $(ab + ba)(aa + bb)^*$
6. $(ab + ba)^*$
7. $(ab + c + b)^*$
8. $(abb + c)^*$
9. $(a + ab)^*(\epsilon + b)$
10. $(abb + b)^*$
11. $((b^*) + (aa)^*)^*$ ◇

Exercise 1.9.19: Building regular expressions. Let $\Sigma = \{a, b\}$. For each of the languages over Σ given below, present a regular expression whose language is the given one.

1. The language containing each string that contains 3 or more a's.
2. The language containing each string that contains 3 or fewer a's.
3. The language containing each string that contains *abba* as a substring.
4. The language containing each string x such that $|x| \geq 4$ and the 4th symbol from the right is b.
5. The language containing each string that contains *baa* as a substring 2 or fewer times.
6. The language containing each string that does not contain two consecutive a's.
7. The language containing each string that begins or ends with *abba*.
8. The language containing each string having even length.
9. $L(b^*a^*) \cap L(a^*b^*)$.
10. $L(b^*a^*b^*) \cap L(a^*b^*a^*)$.
11. $L(b^*ab^*) \cap L(a^*ba^*)$. ◇

Exercise 1.9.20. The DFA of Example 1.1.3 can be viewed as the DFA that results by applying the product construction, given at the beginning of Section 1.2.2, to two DFA. Which two DFA? ◇

Exercise 1.9.21: Unary alphabets and ultimate periodicity. This exercise characterizes the structure of regular languages over a unary alphabet. Define a subset $S \subseteq \mathbb{N}$ of the natural numbers to be **ultimately periodic** if there exist numbers $n \geq 0$ and $p > 0$ such that for all $m \geq n$, it holds that $m \in S$ if and only if $m + p \in S$.

Let Σ be the unary alphabet $\{a\}$. Prove that a language B over Σ is regular if and only if the set $\{n \in \mathbb{N} \mid a^n \in B\}$ is ultimately periodic. ◇

Exercise 1.9.22. Prove that if B is a regular language over Σ, then for any $a \in \Sigma$, the language $B' = \{x \mid xa \in B\}$ is also regular. Prove this by first assuming that M is a DFA with $L(M) = B$, and then showing how to construct a DFA M' with $L(M') = B'$ and whose set of states is equal to that of M. ◇

Exercise 1.9.23. Prove that if B is a regular language over Σ, then the language

$$P = \left\{x \in \Sigma^* \;\middle|\; \exists v \in \Sigma^* \text{ such that } xv \in B\right\}$$

is also regular. The language P contains each prefix of each string in B. ◇

Exercise 1.9.24. Prove that if B is a regular language over Σ, then the language

$$B' = \left\{x \in \Sigma^* \;\middle|\; xx \in B\right\}$$

is also regular. Hint: let $M = (Q, \Sigma, s, T, \delta)$ be a DFA with $L(M) = B$; it may be useful to consider the functions $h_a \colon Q \to Q$, defined for each $a \in \Sigma$, by $h_a(q) = \delta(q, a)$. ◇

Exercise 1.9.25. For each language B over Σ, define

$$\text{FirstHalf}(B) = \left\{x \in \Sigma^* \;\middle|\; \exists y \in \Sigma^* \text{ such that } xy \in B \text{ and } |x| = |y|\right\}.$$

That is, FirstHalf(B) contains the first half of every even-length string in B. Prove that if B is a regular language, then FirstHalf(B) is also regular. ◇

Exercise 1.9.26. Let D denote the language $\{a^m b^n \mid 0 \le n < m\}$. Answer each of the following questions, and justify your answer.

1. Does the pair of strings (a, aa) have a separator, with respect to D?
2. Does the pair of strings (ab, abb) have a separator, with respect to D? ◇

Exercise 1.9.27: Proving non-regularity. For each of the given languages over the alphabet $\Sigma = \{a, b\}$, prove that the language is not regular.

1. The language $\left\{a^i b^j \mid i, j \ge 0, i \ne j\right\}$.
2. The language $\left\{a^i b^j \mid i \ge 0, i^2 = j\right\}$.
3. The language containing each string x such that $\#_a(x) \le \#_b(x)$.
4. The language containing each string x such that $2 \cdot \#_a(x) = 3 \cdot \#_b(x)$.
5. The language containing each string x such that $\#_a(x) \ge 2^{\#_b(x)}$.
6. The language containing each string over Σ whose length is a square, that is, whose length has the form n^2 for a natural number $n \ge 0$.
7. The language containing each string over Σ whose length is a power of 2, that is, whose length has the form 2^n for a natural number $n \ge 0$.
8. The language $\{xx \mid x \in \Sigma^*\}$. (It may be didactic to compare this language with that of Exercise 1.9.24.) ◇

Exercise 1.9.28: Deciding regularity. For each of the following languages over the alphabet $\Sigma = \{a, b\}$, state whether or not the language is regular, and prove your assertion.

1. The language $\left\{a^i b^j \mid i, j \ge 0 \text{ and } (i + j) \text{ is even}\right\}$.
2. The language B that contains each string x such that the number of occurrences of ab as a substring in x is one more than the number of occurrences of ba as a substring in x. As examples, $ab \in B$ and $abba \notin B$.
3. The language C that contains each string x such that the number of occurrences of abb as a substring in x is one more than the number of occurrences of baa as a substring in x. As examples, $ab \notin C$ and $abba \in C$. ◇

Exercise 1.9.29. Let B be the language containing each string over $\{a, b\}$ that has $babbab$ as a substring. Show that any DFA whose language is B must have 7 or more states. ◇

Exercise 1.9.30: Multiples of 7. Let B_7 be the language containing each string over the alphabet $\Sigma = \{0, 1, \ldots, 9\}$ that represents a number (base 10) that is a multiple of 7. Leading zeroes should be ignored, in the sense that a string x is in B_7 if and only if $0x$ is in B_7. Hence, we have $0, 007, 14, 014, 42, 217, 1729 \in B_7$, and we have $6, 06, 10, 237 \notin B_7$.

Give a DFA whose language is B_7. (A diagram is not necessary; the DFA may be specified in any way.) ◇

Exercise 1.9.31: Multiples of 99. Let B_{99} be the language containing each string over the alphabet $\Sigma = \{0, 1, \dots, 9\}$ that represents a number (base 10) that is a multiple of 99. (As in Exercise 1.9.30, leading zeroes should be ignored.) Prove that any DFA whose language is B_{99} must contain 99 or more states. ◇

Exercise 1.9.32: Optimality of the product construction. For each pair of natural numbers $k, \ell \geq 1$, give a DFA M having k states, a DFA M' having ℓ states, and a proof that any DFA whose language is $L(M) \cap L(M')$ must have at least $k \cdot \ell$ states. In a sense, this exercise witnesses the optimality of the product construction of Theorem 1.2.2; this theorem established closure under intersection, for the regular languages. ◇

Exercise 1.9.33: A product construction for NFA. Show how to directly construct, from two NFA $M_1 = (Q_1, \Sigma, \Delta_1, S_1, T_1)$ and $M_2 = (Q_2, \Sigma, \Delta_2, S_2, T_2)$, a third NFA M with state set $Q_1 \times Q_2$ such that $L(M) = L(M_1) \cap L(M_2)$, and prove that your construction works. ◇

Exercise 1.9.34. Consider the ϵ-NFA with state set $Q = \{1, 2\}$, input alphabet $\Sigma = \{a, b\}$, initial state set $S = \{1\}$, accept state set $T = \{1\}$, and the following transition function:

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

Use the subset construction of Theorem 1.3.20 to convert the given ϵ-NFA to a DFA. The resulting DFA will have the same language as the given ϵ-NFA. ◇

Exercise 1.9.35: Reversing regular languages. Prove that if B is a regular language, then the language

$$B' = \{\text{rev}(x) \mid x \in B\}$$

is also regular. Here, rev(x) denotes the reversal of the string x, as in Example 1.6.10. That is, prove that if one starts from a regular language and reverses every single one of its strings, the resulting language is also regular. Hint: try using Theorem 1.5.8. ◇

Exercise 1.9.36: String homomorphisms. Let Σ and Γ be alphabets. For each mapping of the form $h\colon \Sigma \to \Gamma^*$, define the mapping $\overline{h}\colon \Sigma^* \to \Gamma^*$ by $\overline{h}(x_1 \dots x_n) = h(x_1) \cdots h(x_n)$. Each mapping $\overline{h}\colon \Sigma^* \to \Gamma^*$ that arises from a mapping $h\colon \Sigma \to \Gamma^*$ in this way is called a **string homomorphism**. Prove the following statements.

1. If $h\colon \Sigma \to \Gamma^*$ is any mapping, and B is a regular language over Σ, then the language $\{\overline{h}(x) \mid x \in B\}$ is also regular. This result is typically referred to as *closure under homomorphism*.

2. If $h\colon \Sigma \to \Gamma^*$ is any mapping, and C is a regular language over Γ, then the language $\{x \in \Sigma^* \mid \overline{h}(x) \in C\}$ is also regular. This result is typically referred to as *closure under inverse homomorphism*. ◇

Exercise 1.9.37: Rotating regular languages. When B is a language over alphabet Σ, define the *language of rotations of B* as

$$\text{Rotations}(B) = \{yx \mid x, y \in \Sigma^* \text{ and } xy \in B\}.$$

Prove that if B is regular, then Rotations(B) is also regular. ◇

Exercise 1.9.38: Myhill-Nerode relations. Let B be a language over alphabet Σ. Define a **Myhill-Nerode relation** for B to be an equivalence relation $\approx$ on Σ^* that has finite index, refines B, and is a **right congruence** in that, for all strings $x, y \in \Sigma^*$ and for each $a \in \Sigma$, it holds that $x \approx y$ implies $xa \approx ya$. Prove that a language B is regular if and only if there exists a Myhill-Nerode relation for B. ◇

Note 1.9.39. One way to prove the forward direction of Exercise 1.9.38 is by invoking Theorem 1.7.4 and by confirming that $\sim^B$ is a Myhill-Nerode relation. This exercise can thus be interpreted as showing that the notion of *Myhill-Nerode relation* abstracts out the vital properties of $\sim^B$ that ensure regularity of B. ◇

Exercise 1.9.40: The pumping lemma. Prove the *pumping lemma*, which is the following statement. Suppose that B is a regular language over Σ; then, there exists a natural number $K \geq 1$ such that for any string $w \in B$ with $|w| \geq K$, there exist $x, y, z \in \Sigma^*$ such that the following hold:

- $w = xyz$,
- $|xy| \leq K$,
- $y \neq \epsilon$, and
- for all $n \geq 0$, it holds that $xy^nz \in B$.

The pumping lemma gives a necessary condition for regularity. Its contrapositive form thus gives a sufficient condition for non-regularity, and indeed a typical use of this lemma is to show non-regularity of a language via the contrapositive form.

As a hint sketch, this lemma can be proved along the following lines. Let $M = (Q, \Sigma, s, T, \delta)$ be a DFA whose language is B, and set $K = |Q|$. Let $w = w_1 \ldots w_m$ be a string of length $m \geq K$, set $q_0 = s$, and let $q_1, \ldots, q_m$ be the states that the DFA passes through upon processing w, that is, the states such that

$$[q_0, w_1 \ldots w_m] \vdash_M [q_1, w_2 \ldots w_m] \vdash_M [q_2, w_3 \ldots w_m] \vdash_M \cdots \vdash_M [q_m, \epsilon].$$

Then, exploit the fact that two of the $K + 1$ states in the list $q_0, q_1, \ldots, q_K$ must be equal. ◇

Exercise 1.9.41: Non-regularity of the primes. Let B be the language $\{1^n \mid n \text{ is a prime}\}$ containing the primes in unary representation. Prove, using the pumping lemma (of Exercise 1.9.40), that the language B is not regular. ◇

Exercise 1.9.42: Between one string and infinitely many. Let M be a DFA with state set Q. Prove that the language $L(M)$ is infinite if and only if there exists a string $x \in L(M)$ such that the inequalities $|Q| \leq |x| \leq 2|Q| - 1$ hold. ◇

Exercise 1.9.43: The tip of the pyramid. Let Σ be the alphabet $\{r, g, b\}$, whose symbols represent the colors *red*, *green*, and *blue*. Define $d\colon \Sigma^* \setminus (\{\epsilon\} \cup \Sigma) \to \Sigma^*$ as the function that, on a non-empty string $x_1 \ldots x_m$ of length $m \geq 2$, returns the string $y_1 \ldots y_{m-1}$ where, for each $i = 1, \ldots, m-1$, the symbol y_i is defined as x_i if $x_i = x_{i+1}$, and as the unique element of $\Sigma \setminus \{x_i, x_{i+1}\}$ if $x_i \neq x_{i+1}$. That is, the color y_i is derived from the colors x_i and x_{i+1} as follows: if the colors x_i and x_{i+1} are the same, then the color y_i is set equal to them; if the colors x_i and x_{i+1} are different, then the color y_i is set to the unique color that is different from each of them. Define $d^+ : \Sigma^* \setminus \{\epsilon\} \to \Sigma$ as the function that, on a non-empty string x, returns the symbol obtained by applying the function d to x repeatedly, a total of $|x| - 1$ many times. In other words, when x is a non-empty string, $d^+(x)$ is defined as the symbol that results from applying d repeatedly to x until a single symbol remains.

Prove or disprove: the language $B = \{x \in \Sigma^* \setminus \{\epsilon\} \mid d^+(x) = b\}$ is regular. ◇

Exercise 1.9.44: Bisimulations. Let $M = (Q, \Sigma, s, T, \delta)$, $M' = (Q', \Sigma, s', T', \delta')$ be DFA sharing the same input alphabet Σ. Define a **bisimulation** between M and M' as a relation $R \subseteq Q \times Q'$ where, for each pair $(q, q') \in R$, the following hold: $q \in T \Leftrightarrow q' \in T'$, and for each $a \in \Sigma$, the pair $(\delta(q, a), \delta'(q', a))$ is in R. Note that this definition does not depend on the start states of the DFA.

Say that two states $q \in Q$, $q' \in Q'$ are **bisimilar** if there exists a bisimulation between M and M' of which the pair (q, q') is an element. Let us presuppose that what one can observe about a state is whether or not it is accepting, and that one can also subject a state to a transition, based on a given symbol. Then, bisimilar states can be described as being *observationally indistinguishable*: they produce the same observations, and subjecting them to transitions based on a common symbol leads to states that are again indistinguishable.

1. Prove that there exists a bisimulation R between M and M' such that $(s, s') \in R$ if and only if $L(M) = L(M')$.
2. Let $g\colon Q \to Q'$ be a map, and let $R_g = \{(q, g(q)) \mid q \in Q\}$ be its graph. Prove that g is a homomorphism from M to M' if and only if R_g is a bisimulation between M and M' such that $(s, s') \in R_g$.
3. Prove that when R_1, R_2 are bisimulations between M and M', their union $R_1 \cup R_2$ is also a bisimulation between M and M'. This result implies that, when looking at the

bisimulations between two DFA, there is a *greatest bisimulation*—namely, the union of all such bisimulations.

4. Let $\approx_M$ be the **bisimilarity relation** on M, defined as the set containing each pair of states $(p, q) \in Q \times Q$ such that there exists a bisimulation between M and itself of which the pair (p, q) is an element. This relation is clearly the greatest bisimulation between M and itself, in the just-introduced sense. Prove that the relation $\approx_M$ is equal to the relation $\overline{R}$ treated by Lemma 1.8.1.
5. By modifying the *marking phase* in Section 1.8, give an algorithm for computing the complement of the greatest bisimulation between M and M'; here, the complement is with respect to the set $Q \times Q'$. The resulting algorithm allows us to determine whether or not M and M' are equivalent in the sense of having the same language: by part 1 of this exercise, we have that $L(M) = L(M')$ if and only if the pair (s, s') is not in this complement. Hint: first mark the states in $T \times (Q' \setminus T')$ and in $(Q \setminus T) \times T'$. ◇

1.10 Bibliographic discussion

General references on the theory of computation include the books by Hopcroft, Motwani, and Ullman (2007); Kozen (1997, 2006); Moore and Mertens (2011); Papadimitriou (1994); and Sipser (2013).

An early study of finite-state systems was conducted in an article of McCulloch and Pitts (1943). In the 1950s, versions of the DFA model were presented and studied (Huffman 1954; Mealy 1955; Moore 1956; Kleene 1956). The NFA model is due to an article of Rabin and Scott (1959), who established the equivalence to the DFA model, in the sense of Theorem 1.3.24. Closure properties of the regular languages were studied by many authors, including Kleene (1956); Ginsburg and Rose (1963); and, Rabin and Scott (1959). The characterization of regular languages via regular expressions given in Theorem 1.5.8 is due to Kleene (1956); our presentation of this result is based on that of Kozen (1997). The Myhill-Nerode theory in and around Section 1.7, and in Exercise 1.9.38, is due to Myhill (1957) and Nerode (1958). DFA minimization procedures were studied by numerous authors, including Huffman (1954), Moore (1956), Nerode (1958), and Hopcroft (1971).

Our discussion in Example 1.5.7 of the language of *alternating strings* stems from a textbook discussion of this language (Hopcroft, Motwani, and Ullman 2007, Chapter 3). The pumping lemma of Exercise 1.9.40 is due to Bar-Hillel, Perles, and Shamir (1961). Exercise 1.9.44 is based on an article of Rutten (1998).

2 Computability Theory

I wonder why. I wonder why.
I wonder why I wonder.
I wonder why I wonder why
I wonder why I wonder!
— Richard Feynman

The problem with introspection is that it has no end.
— Philip K. Dick, *The Transmigration of Timothy Archer*

The finite-state automata of the previous chapter were relatively limited computational models: an automaton could only make one pass through an input string, and had no working space, apart from its bounded memory. This chapter turns to study *Turing machines*, computational models which are considerably more general and more powerful than automata, and which indeed are the most powerful models that we will study. A particular type of Turing machine, the *halting deterministic Turing machine*, will be presented as a formalization of the intuitive notion of *algorithm*; the corresponding class of languages that these machines define are called the *computable languages*. Just as the previous chapter explored both the scope and the boundaries of the regular languages, this chapter engages in a kindred exploration of the computable languages.

2.1 Deterministic Turing machines

2.1.1 Introduction

The notion of *algorithm* is an informal and intuitive one; an early known example of an algorithm, from ancient Greece, is *Euclid's algorithm* for computing the greatest common divisor of two positive natural numbers. By an **algorithm**, we here refer to a procedure with the following properties. An algorithm is specified via a finite list of instructions; each instruction is finite and unambiguously describes an action performable mechanically, without recourse to judgement or creativity. An algorithm operates deterministically, and in discrete time steps. When executed on an input, an algorithm terminates after a finite number of steps, producing the desired output.

This section presents the *deterministic Turing machine (DTM)*, a general-purpose, full-fledged computational model that is recognized as providing a formalization of the intuitive notion of *algorithm*, and thus as capturing the concept of computation in a broad sense. The DTM is a mathematical object, a construct for performing analysis—as was the case for the DFA and the other automata models previously seen. The definition of DTM will naturally lead us to define the notion of a *computable* language, which amounts to a language for which there exists an algorithm determining membership. In this chapter, the quantity of time and space consumed during a computation is not in any way restricted, for the focus is on the power of computation *in principle*.

The DTM was introduced in a 1937 publication by Alan Turing. Around this time, there were in fact numerous proposals of computational models, in addition to the DTM, that aimed to formalize the notion of algorithm. These proposals stemmed from mathematical questions of the era about whether certain problems were solvable by algorithms; in order to rigorously address these questions, a precise definition of algorithm was needed. At the time, the proposed computational models seemed quite qualitatively different from each other; for example, one was based on functions defined on the natural numbers, and another, the λ-*calculus*, was based on a simple, abstract view of function formation and application. Remarkably, these differences were revealed to be superficial: all of these computational models turned out to be provably equivalent, in that membership in a language could be computed by one model if and only if it could be computed by another. Typical programming languages used in practice, when formalized, also yield computational models that are equivalent to the original ones. This rich system of equivalences signals the robustness and the stability of the notion of *computable* language. These equivalences back the non-mathematical claim that each of these models provides a suitable formalization of the intuitive notion of algorithm—a claim known as the *Church-Turing thesis*, and credited to the 1937 article of Turing and a 1936 article of Alonzo Church.

The situation of having a single mathematical notion with an abundance of seemingly disparate characterizations strongly suggests that this notion is natural and primal. But, if there are numerous computational models that formalize the notion of algorithm, why do we focus on the DTM in our study? A prime reason is the *simplicity* of the DTM model. It can be presented with relative ease (particularly when the DFA model has been understood), and its stripped-down nature facilitates proving the types of results that are of interest to us. Furthermore, the DTM model makes it easy and natural to impose time or space bounds on computation; such bounds are central to the study of complexity theory. What is lost and traded off by considering such a simple model is that it can be cumbersome to precisely present sophisticated algorithms. However, the precise presentation of algorithms by DTMs is not our focus, and we typically specify algorithms informally, appealing to the reader's sense that they could be implemented by DTMs—if laboriously.

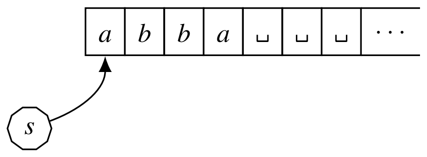


Figure 2.1.1. The initial configuration of a DTM on the input string *abba*. The *control* is depicted by a polygon enclosing the current state, which in this case is the DTM's start state s. The *head* is depicted by the tip of an arrow that emanates out of the control. The *worktape* here contains the string *abba* in its initial 4 cells, and the *blank symbol* in all other cells.

2.1.2 Model

We begin with an overview of the model. A deterministic Turing machine (DTM) has a one-dimensional working space, its *worktape*. The input string of a DTM is presented on this tape, with each symbol in a memory unit; beyond the input string, this tape has further memory units, each capable of holding one symbol, with which the DTM can compute. While a DTM accesses its tape via a *head* that is located at and can operate on one memory unit at a time, this head can move in both directions, and can write symbols in addition to reading them. A more comprehensive, informal description of the DTM model follows.

Architecture-wise, a DTM consists of a **control**, a **head**, and a **worktape**. The worktape constitutes the working space of a DTM; it is a one-dimensional array, which we conceive of as horizontally positioned. The worktape is infinite to the right, but has a left end; it consists of discrete memory units called **cells**, each of which can store one symbol. At any point in time, the control is in a state, and the head is located at a single worktape cell. A DTM can move its head both to the left and right during a computation, and can both read from and write to the worktape via its head; this is in contrast to a DFA, which scans its input just once, from left to right, in a read-only fashion.

Given an input, a DTM begins in the configuration where the control is in the DTM's *start state*; the head is at the leftmost cell of the worktape; and the worktape contains the input in its initial cells but otherwise contains a special symbol, the **blank symbol** ␣. Figure 2.1.1 illustrates an initial configuration of a DTM.

A DTM operates in discrete time steps, as with a DFA. In each time step, a DTM makes a transition based on its current state and the symbol in the cell where its head is located. To make a transition, a DTM performs three changes:

- It changes state.
- It writes a tape symbol at the location of its head.
- It moves its head left or right.

Correspondingly, a DTM's *transition function* needs to specify, for each state and symbol, three pieces of information: a state, a symbol, and a direction—left or right, to be represented by -1 and $+1$, respectively. Recall that a DFA made a transition simply by changing state; correspondingly, its transition function specified, for each state and symbol, just one piece of information, a state.

When run on an input, a DTM makes transitions until it enters its *accept state* or its *reject state*, at which point it comes to rest and ceases to make transitions; so, formally, the transition function is not defined on these two states. It is certainly possible that, when invoked on an input, a DTM will never enter its accept state or its reject state, but rather, runs infinitely without *halting*—in this case, we will say that the DTM *loops* on the input.

We next turn to present the formal definition of a DTM.

Definition 2.1.1. A **deterministic Turing machine (DTM)** is a 7-tuple $(Q, \Sigma, \Gamma, s, t, r, \delta)$ where:

- Q is a finite set called the **state set**,
- Σ is an alphabet called the **input alphabet**,
- Γ is an alphabet called the **tape alphabet** and is such that $\Gamma \supseteq \Sigma$,
- $s \in Q$ is a state called the **start state** or **initial state**,
- $t \in Q$ is a state called the **accept state**,
- $r \in Q$ is a state called the **reject state** and is such that $r \neq t$, and
- $\delta \colon (Q \setminus \{t, r\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ is a function called the **transition function**.

It is required that the *blank symbol* $\llcorner\!\lrcorner$ is an element of $\Gamma \setminus \Sigma$. ◇

Let us remark that there are multiple ways to define the DTM; here, we elected a definition of minimalist design.

As mentioned, when a DTM is invoked on a particular input string, its tape is initialized to contain the string at the left end, followed by an infinite sequence of blank symbols; and the DTM's head is located at the leftmost position of the tape. Each input string is required to be over the input alphabet Σ; the assumption that the blank symbol is not in Σ permits the DTM to detect where the input string ends. In carrying out a computation, a DTM may write elements from the tape alphabet Γ onto the tape, and its transition function must be fully defined on all pairs consisting of a state (that is not t nor r) and a symbol from Γ.

We next formalize the notion of a *configuration* of a DTM. Recall that a configuration ought to contain all of the information that one needs to know, at a particular point in time, about how a computation will proceed. For a DTM, then, a configuration will provide the state of its control, the entire contents of its tape, and the location of its head. We assume that the worktape cells are numbered with indices, starting from the left end, by $1, 2, 3, \ldots$; this is depicted in Figure 2.1.2. So, a head location can be specified as an element of $\mathbb{N}^+$; and, the tape contents can be given by a function that maps a cell's number to the symbol that it contains, that is, by a function from the set $\mathbb{N}^+$ to the tape alphabet Γ.

1	2	3	4	5	6	7	
a	b	b	a	$\sqcup$	$\sqcup$	$\sqcup$	$\cdots$

Figure 2.1.2. A DTM's worktape cells are numbered with indices, starting from 1 on the left. The example tape contents given here are represented by a function $\tau\colon \mathbb{N}^+ \to \Gamma$ where $\tau(1) = a$, $\tau(2) = b$, $\tau(3) = b$, $\tau(4) = a$, and $\tau(i)$ is the blank symbol for all $i \geq 5$.

We also next give the definition of *successor configuration* of a configuration. To define this notion comprehensively, it is necessary to specify what occurs if a DTM tries to move its head left when it is at the leftmost cell; the convention we use is that, in this case, the DTM's head remains at the leftmost cell. Hence, if the DTM's head is at location $\ell \in \mathbb{N}^+$ and a transition calls for the head to move in the direction $d \in \{-1, +1\}$, the next location will be $\ell + d$ unless this sum is 0, in which case the next location should be 1; this value is expressed below as $\max(\ell + d, 1)$.

Definition 2.1.2. Let $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ be a DTM.

- A **configuration** of M is a triple $[q, \tau, \ell]$ where $q \in Q$ is a state, τ is a function from $\mathbb{N}^+$ to Γ representing tape contents, and $\ell \in \mathbb{N}^+$ is a head location.
- The **successor configuration** of a configuration $[q, \tau, \ell]$ is defined when $q \in Q \setminus \{t, r\}$. In this case, set $(p, a, d) = \delta(q, \tau(\ell))$; then, the successor configuration of $[q, \tau, \ell]$ is defined as the configuration $[p, \tau[\ell \mapsto a], \max(\ell + d, 1)]$; here, $\tau[\ell \mapsto a]$ denotes the function that maps ℓ to a, and is otherwise equal to τ.

To discuss configurations of a DTM M, we use the previously given definitions of the relations $\vdash_M$, $\vdash_M^n$, and $\vdash_M^*$ presented in Definition 1.1.8. ◇

In order to smoothly present configurations of DTMs, we need a convenient way to specify a function $\tau\colon \mathbb{N}^+ \to \Gamma$ giving the tape contents; such a function is an infinite object. To this end, we view such a function as an infinite string, that is, as the infinite sequence of symbols $\tau(1)\tau(2)\tau(3)\ldots$; and we use the notation $\sqcup\ldots$ to indicate the infinite string consisting solely of blanks. So, for example, we will use the notation $bab\sqcup\ldots$ to denote the function $\tau\colon \mathbb{N}^+ \to \Gamma$ that maps 1 to b, 2 to a, 3 to b, and each other element of $\mathbb{N}^+$ to $\sqcup$; as another example, we will use the notation $abba\sqcup\ldots$ to denote the tape contents and the function given in Figure 2.1.2. (Note that the tape contents, when viewed as an infinite string, will always terminate with an infinite sequence of blanks.)

DTM examples

We next examine two examples of DTMs.

Example 2.1.3. Consider the DTM $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$, presented in Figure 2.1.3, where $Q = \{s, t, r, h, e, g\}$, $\Sigma = \{a, b\}$, and $\Gamma = \{a, b, \sqcup\}$.

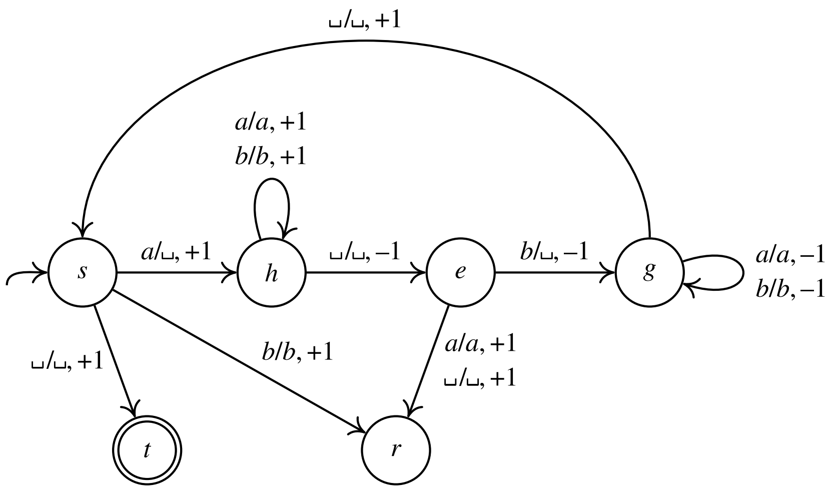


Figure 2.1.3. The DTM given in Example 2.1.3. In general, we form the diagram for a DTM as follows. Each state is placed in a circle; the initial state is indicated by an unlabeled arrow that points to it; the accept state has a double circle placed around it. Each transition $\delta(q, c) = (p, a, d)$ is indicated by an arrow with label $c/a, d$ from the state q to the state p; multiple transitions having the same source and target states are indicated using multiple labels on a shared arrow. Thus, to determine the behavior of the transition function δ given a state q and a read symbol c, one looks for an arrow coming out of state q having a label whose first component is c; the label's remaining components specify the symbol to be written and the direction of movement, and the arrow's target is the state to be entered. The accept and reject states are the only states with no outgoing arrows.

This DTM's transition function δ is given by the following table:

δ	a	b	$\sqcup$
s	$(h, \sqcup, +1)$	$(r, b, +1)$	$(t, \sqcup, +1)$
h	$(h, a, +1)$	$(h, b, +1)$	$(e, \sqcup, -1)$
e	$(r, a, +1)$	$(g, \sqcup, -1)$	$(r, \sqcup, +1)$
g	$(g, a, -1)$	$(g, b, -1)$	$(s, \sqcup, +1)$

On the input string a, this DTM begins in the initial configuration $[s, a\sqcup\ldots, 1]$. We have

$$\begin{aligned}[s, a\sqcup\ldots, 1] &\vdash_M [h, \sqcup\sqcup\ldots, 2]\\ &\vdash_M [e, \sqcup\sqcup\ldots, 1]\\ &\vdash_M [r, \sqcup\sqcup\ldots, 2].\end{aligned}$$

Once in the configuration $[r, \sqcup\sqcup\ldots, 2]$, the machine halts in its reject state r. The string a is considered *rejected*. ◇

Remark 2.1.4. Before proceeding further, let us add a detail to the fashion in which we present configurations. To enhance readability, when displaying concrete configurations, we will typically underline the symbol in the tape string where the head is located. This convention should facilitate determining successor configurations of DTMs. ◇

As in the previous chapter, we use the term **computation** to refer to a sequence of all configurations that a machine passes through when invoked on an input string.

Example 2.1.5. Having put the convention of Remark 2.1.4 in effect, we exhibit further computations of the DTM M from Example 2.1.3.

On the input string ab, this DTM begins in the initial configuration $[s, \underline{a}b\sqcup\sqcup\ldots, 1]$. We have the computation

$$\begin{aligned}
[s, \underline{a}b\sqcup\sqcup\ldots, 1] &\vdash_M [h, \sqcup\underline{b}\sqcup\sqcup\ldots, 2]\\
&\vdash_M [h, \sqcup b\underline{\sqcup}\sqcup\ldots, 3]\\
&\vdash_M [e, \sqcup\underline{b}\sqcup\sqcup\ldots, 2]\\
&\vdash_M [g, \underline{\sqcup}\sqcup\sqcup\sqcup\ldots, 1]\\
&\vdash_M [s, \sqcup\underline{\sqcup}\sqcup\sqcup\ldots, 2]\\
&\vdash_M [t, \sqcup\sqcup\underline{\sqcup}\sqcup\ldots, 3].
\end{aligned}$$

Once in the configuration $[t, \sqcup\sqcup\underline{\sqcup}\sqcup\ldots, 3]$, the machine halts, in its accept state t; the string ab is considered *accepted.*

On the input string abb, this DTM M begins in the initial configuration $[s, \underline{a}bb\sqcup\sqcup\ldots, 1]$. We have the computation

$$\begin{aligned}
[s, \underline{a}bb\sqcup\sqcup\ldots, 1] &\vdash_M [h, \sqcup\underline{b}b\sqcup\sqcup\ldots, 2]\\
&\vdash_M [h, \sqcup b\underline{b}\sqcup\sqcup\ldots, 3]\\
&\vdash_M [h, \sqcup bb\underline{\sqcup}\sqcup\ldots, 4]\\
&\vdash_M [e, \sqcup b\underline{b}\sqcup\sqcup\ldots, 3]\\
&\vdash_M [g, \sqcup\underline{b}\sqcup\sqcup\sqcup\ldots, 2]\\
&\vdash_M [g, \underline{\sqcup}b\sqcup\sqcup\sqcup\ldots, 1]\\
&\vdash_M [s, \sqcup\underline{b}\sqcup\sqcup\sqcup\ldots, 2]\\
&\vdash_M [r, \sqcup b\underline{\sqcup}\sqcup\sqcup\ldots, 3].
\end{aligned}$$

Here, the DTM halts in its reject state, so the string abb is *rejected* by this DTM. ◇

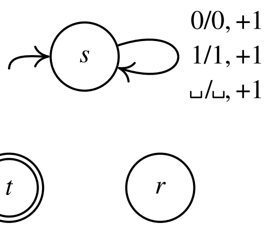


Figure 2.1.4. The DTM N given in Example 2.1.6. The diagram is formed from the specification of N in the manner described by Figure 2.1.3. Neither the accept state nor the reject state is ever entered by this DTM, and so this DTM does not halt on any input.

Example 2.1.6 (A runaway DTM). Let us give, as another example, a tiny DTM. Consider the DTM $N = (Q, \Sigma, \Gamma, s, t, r, \delta)$, presented in Figure 2.1.4, where $Q = \{s, t, r\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \sqcup\}$, and δ is given by the following table:

δ	0	1	$\sqcup$
s	$(s, 0, +1)$	$(s, 1, +1)$	$(s, \sqcup, +1)$

At each time step, this DTM simply moves to the right, mindlessly! For example, on the input string 10, we have the computation

$$[s, \underline{1}0\sqcup\sqcup\sqcup\ldots, 1] \vdash_N [s, 1\underline{0}\sqcup\sqcup\sqcup\ldots, 2] \vdash_N [s, 10\underline{\sqcup}\sqcup\sqcup\ldots, 3] \vdash_N [s, 10\sqcup\underline{\sqcup}\sqcup\ldots, 4] \vdash_N \cdots.$$

In particular, this DTM never enters its accept state or its reject state, and thus does not accept or reject any string. To make use of terminology formalized below, on every input string $x \in \Sigma^*$, this DTM does not *halt*, but *loops*. ◇

Outcomes

Let us define precisely the outcomes possible when a DTM is run on an input string. Let $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ be a DTM.

Definition 2.1.7. Define the **initial configuration** of M on a string $x \in \Sigma^*$ as the configuration $[s, x\sqcup\ldots, 1]$. ◇

Definition 2.1.8. We say that a configuration $[q, \tau, \ell]$ of M is

- an **accepting configuration** if $q = t$;
- a **rejecting configuration** if $q = r$; and,
- a **halting configuration** if $q \in \{t, r\}$, that is, if it is either accepting or rejecting. ◇

Definition 2.1.9. Let $x \in \Sigma^*$ be a string; let α_x be the initial configuration of M on x.

- We say that M **accepts** x if there exists an accepting configuration β such that $\alpha_x \vdash_M^* \beta$.
- We say that M **rejects** x if there exists a rejecting configuration β such that $\alpha_x \vdash_M^* \beta$.

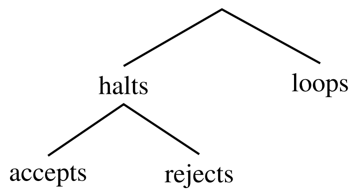


Figure 2.1.5. Tree showing the possible outcomes when a DTM is run on an input string.

- In the case that M accepts or rejects x, we say that M **halts** on x; otherwise, we say that M **loops** on x. ◇

Definition 2.1.10. We define the **language** of a DTM M, denoted by $L(M)$, to be the set $\{x \in \Sigma^* \mid M \text{ accepts } x\}$. ◇

Let us offer some remarks on the introduced definitions. During a computation, once a DTM enters a halting configuration, it ceases to make transitions; a configuration that is halting is (by definition) either accepting or rejecting, but cannot be both, by the proviso that $r \neq t$ (in Definition 2.1.1). So, on an input, a DTM either halts or loops, but not both; if it halts, it either accepts or rejects, but not both. Consequently, we can observe that, when a DTM is run on an input, exactly one of three outcomes occurs: the DTM accepts, rejects, or loops. (This is a point of contrast with the DFA model; recall that when a DFA is run on an input, exactly one of two outcomes occurs: the DFA either accepts or rejects.) These three DTM outcomes are depicted in Figure 2.1.5. Thus, when a string y is in the language $L(M)$ of a DTM M, it holds that M accepts y; and when a string y is not in the language $L(M)$ of a DTM M, what can be generally inferred is that M either rejects y or loops on y.

Let us emphasize that we here use the word *loops* in a specific terminological fashion: a DTM is said to *loop* on a string when it does not halt on the string. When a DTM loops on a string in this sense, it is not necessarily the case that the DTM's computation on the string has a configuration that appears more than once. (For example, when invoked on any input string, the DTM of Example 2.1.6 never repeats configuration.)

Example 2.1.11 (Continuation of Example 2.1.3). The DTM M of Example 2.1.3 halts on each input, and its language $L(M)$ is in fact equal to a key language previously seen (in Section 1.6): the language $E = \{a^n b^n \mid n \geq 0\}$. Let us explain why.

On a high level, when invoked in the state s, this DTM attempts to remove an instance of the symbol a from the left boundary of the string that starts from the head location; to then remove an instance of the symbol b from the right boundary of this string; and then to move back to the new left boundary and to iterate this process.

More concretely, let us assume that the DTM is invoked in state s and at a location such that, beginning from the location and continuing to the right, the tape consists of a string in $\{a, b\}^*$ followed by blank symbols. The DTM rejects if the symbol scanned is b, and accepts if the symbol scanned is the blank symbol. (This act of acceptance is consistent with the desire to accept the strings in the language E: we have that the empty string ϵ is included in E.) If the symbol scanned is a, this symbol is overwritten with the blank symbol, and the DTM changes to state h. In state h, one can then think of the DTM as **h**olding an instance of the symbol a.

Once in state h, the DTM iteratively moves to the right and stays in h so long as it sees the symbol a or the symbol b. When it encounters a blank, it transitions to state e, wherein it is ready to **e**at an instance of the symbol b. After first transitioning into state e, the DTM's head is located at the last non-blank symbol—if this exists. If the symbol at the head is not b, then the DTM rejects; if it is, then the DTM transitions into the state g, overwrites the b with a blank symbol (in effect, eating the b), and steps to the left.

In the state g, the DTM tries to **g**o to the leftmost boundary of the string; it does this by staying in the state g until it encounters a blank symbol. Note that in the configuration where the state g is first entered, there is either a blank at the head's location or to the left of the head, since a blank was written when the DTM most recently came out of the s state. When a blank is encountered in state g, the DTM transitions to state s and moves to the right, and hence iterates the just-described process on a shorter string in $\{a, b\}^*$. ◇

Further DTM examples

We next give two more examples of DTMs.

Example 2.1.12. Consider the DTM $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$, presented in Figure 2.1.6, where $Q = \{s, u, t, r\}$, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b, c, \sqcup\}$, and δ is given by the following table:

δ	a	b	c	$\sqcup$
s	$(t, a, +1)$	$(r, b, +1)$	$(u, c, +1)$	$(u, \sqcup, +1)$
u	$(u, a, +1)$	$(u, b, +1)$	$(u, c, +1)$	$(u, \sqcup, +1)$

This DTM accepts immediately if its input begins with the symbol a; it rejects immediately if its input begins with the symbol b; and otherwise, it enters state u and walks indefinitely to the right. Thus, its language $L(M)$ is the set of all strings over Σ that begin with the symbol a; and this DTM halts on precisely the strings that begin with either the symbol a or the symbol b.

Let us consider some example computations. When invoked on the input string $abba$, this DTM accepts after making one transition:

$$[s, \underline{a}bba\sqcup\ldots, 1] \vdash_M [t, a\underline{b}ba\sqcup\ldots, 2].$$

And, when invoked on the input string ca, this DTM makes a head move to the right at each step, and thus loops:

$$[s, \underline{c}a\sqcup\sqcup\ldots, 1] \vdash_M [u, c\underline{a}\sqcup\sqcup\ldots, 2] \vdash_M [u, ca\underline{\sqcup}\sqcup\ldots, 3] \vdash_M \cdots .$$ ◇

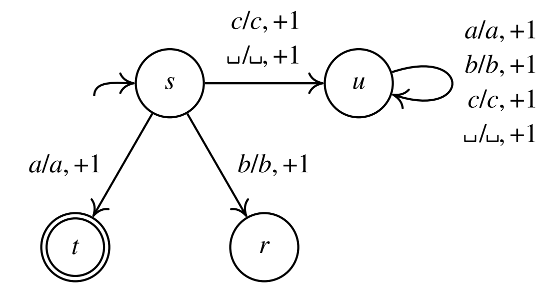


Figure 2.1.6. The DTM given in Example 2.1.12. The diagram is formed using the conventions described in Figure 2.1.3.

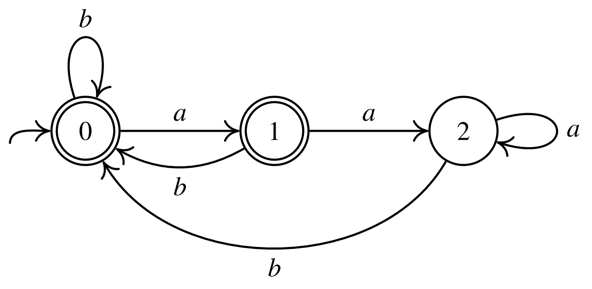


Figure 2.1.7. The DFA discussed in Example 2.1.13.

Example 2.1.13 (A DTM based on a DFA). We revisit the DFA $M = (Q, \Sigma, s, T, \delta)$ of Example 1.1.5. This DFA M, shown in Figure 2.1.7, has $Q = \{0, 1, 2\}$, $\Sigma = \{a, b\}$, $s = 0$, $T = \{0, 1\}$, and transition function δ defined by:

δ	a	b
0	1	0
1	2	0
2	2	0

We here give a DTM M' based on the DFA M. In particular, each input string is accepted or rejected by M' according to whether it is accepted or rejected by M. We define the DTM M', which is displayed in Figure 2.1.8, as $(Q', \Sigma, \Gamma, 0, t, r, \delta')$ where 0 is the start state, $Q' = \{0, 1, 2, t, r\}$, $\Gamma = \{a, b, \sqcup\}$, and δ' is given by the following table:

δ'	a	b	$\sqcup$
0	$(1, a, +1)$	$(0, b, +1)$	$(t, \sqcup, +1)$
1	$(2, a, +1)$	$(0, b, +1)$	$(t, \sqcup, +1)$
2	$(2, a, +1)$	$(0, b, +1)$	$(r, \sqcup, +1)$

It can be seen that the state component of $\delta'(q, d)$, for each $q \in Q$ and $d \in \Sigma$, is equal to the value $\delta(q, d)$ provided by the transition function δ of the DFA M. Indeed, this DTM imitates the behavior of the DFA M: as long as it reads symbols from $\Sigma = \{a, b\}$, it takes steps to the right and makes precisely the state transitions that the DFA M would make.

As an example, let us consider the input string *aaba*. When invoked on this string, the DFA M produces the following computation:

$$\begin{aligned}
[0, aaba] &\vdash_M [1, aba] \\
&\vdash_M [2, ba] \\
&\vdash_M [0, a] \\
&\vdash_M [1, \epsilon].
\end{aligned}$$

When invoked on this string, the DTM M' produces the following computation:

$$\begin{aligned}
[0, \underline{a}aba\sqcup\sqcup\sqcup\ldots, 1] &\vdash_{M'} [1, a\underline{a}ba\sqcup\sqcup\sqcup\ldots, 2] \\
&\vdash_{M'} [2, aa\underline{b}a\sqcup\sqcup\sqcup\ldots, 3] \\
&\vdash_{M'} [0, aab\underline{a}\sqcup\sqcup\sqcup\ldots, 4] \\
&\vdash_{M'} [1, aaba\underline{\sqcup}\sqcup\sqcup\ldots, 5] \\
&\vdash_{M'} [t, aaba\sqcup\underline{\sqcup}\sqcup\ldots, 6].
\end{aligned}$$

Recall that, by definition, a DTM has a unique accept state, while a DFA may have multiple accept states, as is the case for the DFA M considered here. Despite this difference in definition, we have succesfully presented a DTM whose behavior is faithful to that of the DFA M; as soon as the DTM reads a blank symbol, it knows that it has reached the end of the input string and then enters its accept state t or its reject state r depending on whether or not its state (0, 1, or 2) was an accept or reject state of the DFA M, respectively.

In this example, we showed how to imitate a particular DFA by a DTM; the ideas we used to do so can be generalized to show that any DFA can be imitated by a DTM. We formulate this claim below as Proposition 2.1.18; see also the discussion that follows. ◇

2.1.3 Classes of languages

We next define two classes of languages using the DTM model. First and foremost, we define what it means for a language to be *computable*; this definition is intended to formalize what it means for membership in the language to be decidable by an algorithm. Recall that an *algorithm* is presupposed to terminate after a finite number of steps, on any input. Correspondingly, to define the notion of *computable* language, we want to only permit a DTM if it halts after a finite number of steps, on each input; such a DTM is formalized

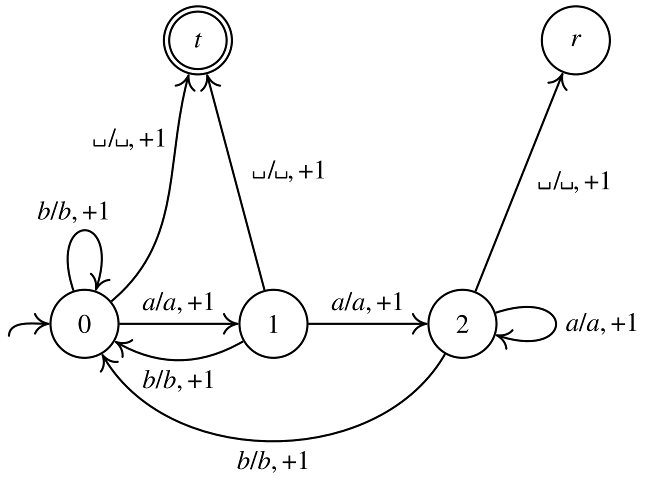


Figure 2.1.8. The DTM given in Example 2.1.13; this DTM is constructed based on the DFA given in Figure 2.1.7. The diagram is formed using the conventions described in Figure 2.1.3.

here as a *halting DTM*. Indeed, imagine that one initiates a DTM computation on an input; the utility of doing this is not at all clear if the computation is not guaranteed to halt.[5]

Definition 2.1.14. A DTM M with input alphabet Σ is called **halting** if M halts on every string $x \in \Sigma^*$. When M is a halting DTM, we say that M **decides** its language $L(M)$. ◇

Definition 2.1.15. A language B is **computable** if there exists a halting DTM M such that $B = L(M)$. ◇

Note that computable languages were historically referred to as *recursive* languages, and are also referred to as *decidable* languages.

Example 2.1.16. Let us examine the four example DTMs in the previous section to see which are halting:

- The DTM of Example 2.1.3 is halting, as discussed in Example 2.1.11.
- The DTM of Example 2.1.6 is not halting; indeed, it does not halt on any input.
- The DTM of Example 2.1.12 does not halt on all inputs; in particular, it does not halt on inputs beginning with the symbol c; thus, this DTM is not halting.
- The DTM of Example 2.1.13 is halting: on any input, it moves to the right until it scans the blank symbol, at which point it halts.

5. Even if there is such a guarantee, the amount of time that the computation will take should also be considered; but that is the concern of the next chapter.

Note that even if a DTM M is not halting, its *language* $L(M)$ may be computable; there may still exist a halting DTM sharing the same language. For example, the DTM N of Example 2.1.6 is not halting, but its language $L(N)$, the empty set $\emptyset$, is computable, via (for example) a DTM that immediately rejects each input string. ◇

This is an opportune moment to define what it means for a *function* to be computable. Essentially, a function $f\colon \Sigma^* \to \Sigma^*$ is defined to be *computable* if there exists a DTM M such that, when M is run on any string $x \in \Sigma^*$, it terminates in an accepting configuration where the tape contains $f(x)$ followed by blank symbols, and the head is at the leftmost location (namely, the location numbered 1).

Definition 2.1.17. A DTM $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ **computes** a function $f\colon \Sigma^* \to \Sigma^*$ if, for each string $x \in \Sigma^*$, it holds that

$$[s, x\sqcup\ldots, 1] \vdash_M^* [t, f(x)\sqcup\ldots, 1].$$

A function $f\colon \Sigma^* \to \Sigma^*$ is **computable** if there exists a DTM that computes it. ◇

We can observe that any DTM computing a function is a halting DTM.

Let us next compare the notions of regular language and computable language. On the one hand, we have the following.

Proposition 2.1.18. *Each language that is regular is also computable.*

This proposition can be proved by showing how to pass from a DFA to a DTM that imitates the behavior of the DFA; this can be done by generalizing the idea of Example 2.1.13, where a concrete DFA was converted to a DTM. We leave the arguing of this proposition as an exercise (Exercise 2.10.13).

The converse of Proposition 2.1.18 does not hold; the following proposition implies that the class of computable languages strictly contains the class of regular languages.

Proposition 2.1.19. *The language* $\{a^n b^n \mid n \geq 0\}$ *is computable but is not regular.*

Proof. By the discussion in Example 2.1.11, the DTM given in Example 2.1.3 is halting, and its language is the specified one. Hence, this language is computable. Example 1.6.9 established that this language is not regular. □

We now present the second class of languages defined in terms of the DTM model. The definition of *computable* language, as discussed, is a formalization of what it means for a language to be computable by an algorithm. In contrast, the class of languages that we next define are presented for the purpose of analysis.

Definition 2.1.20. A language B is **computably enumerable** (for short, **CE**) if there exists a DTM M such that $B = L(M)$. ◇

Computably enumerable languages were historically referred to as *recursively enumerable* languages, and are also referred to as *semi-decidable* languages.

According to the definitions, the requirement for a language to be computably enumerable is clearly a relaxation of the requirement for a language to be computable; the stipulation that the DTM be *halting* is lifted. We record this fact, to be used tacitly in the sequel, as follows.

Proposition 2.1.21. *Each language that is computable is also computably enumerable.*

To recapitulate, a language B is computable if there exists a DTM that accepts each string in B, and rejects each string outside B; a language B is computably enumerable if there exists a DTM that accepts each string in B, and does not accept any string outside B—so, the DTM either rejects or loops on each string outside B. One might succinctly say that the definition of *computable* is based on the distinction between acceptance and rejection, whereas the definition of *computably enumerable* is based on the distinction between acceptance and nonacceptance.

Remark 2.1.22 (On the presentation of DTMs). When we claim the existence of a DTM with a particular behavior, in the sequel we typically do *not* present a DTM formally by giving all parts of the 7-tuple in the definition of DTM, as we did in Examples 2.1.3, 2.1.6, 2.1.12, and 2.1.13. Rather, we give a high-level description of what the DTM should do; we appeal to the reader's sense, intuition, and judgment that the description could be implemented by a DTM (if arduously). Recall that the original impetus behind introducing the DTM was, in any case, to formalize the *intuitive* notion of algorithm. ◇

2.1.4 Summary of models and language classes

The following table shows the computational models that have been studied so far, along with the language classes that they define:

Computational model	Defined class of languages	Justification
DFA	regular languages	Definition 1.1.1
NFA	regular languages	Theorem 1.3.24
ϵ-NFA	regular languages	Theorem 1.3.24
halting DTM	computable languages	Definition 2.1.15
DTM	CE languages	Definition 2.1.20

As discussed in Section 2.1.3, each language that is regular is also computable, but not vice versa; and each language that is computable is also CE. The relationship between the computable languages and the CE languages will be clarified later in this chapter (specifically, in Section 2.5.1).