# Automata and Computability

Fall 2020

Alexis Maciel
Department of Computer Science
Clarkson University

# Contents

# Preface

If you've taken a few computer science courses, you may now have the feeling that programs can always be written to solve any computational problem. Writing the program may be hard work. For example, it may involve learning a difficult technique. And many hours of debugging. But with enough time and effort, the program can be written.

So it may come as a surprise that this is not the case: there are computational problems for which no program exists. And these are not ill-defined problems — Can a computer fall in love? — or uninteresting toy problems. These are precisely defined problems with important practical applications.

Theoretical computer science can be briefly described as the mathematical study of computation. These notes will introduce you to this branch of computer science by focusing on computability theory and automata theory. You will learn how to precisely define what computation is and why certain computational problems cannot be solved. You will also learn several concepts and techniques that have important applications. Chapter 1 provides a more detailed introduction to this rich and beautiful area of computer science.

These notes were written for the course CS345 *Automata Theory and Formal Languages* taught at Clarkson University. The course is also listed as MA345 and CS541. The prerequisites are CS142 (a second course in programming) and

MA211 (a course on discrete mathematics with a focus on writing mathematical proofs).

These notes were typeset using LaTeX (MiKTeX implementation with the TeXworks environment). The paper size and margins are set small to make it easier to read the notes on the relatively small screen of a laptop or tablet. But then, if you're going to print the notes, don't print them to "fit the page". That would give you pages with huge text and tiny margins. Instead, print them double-sided, at "actual size" (100%) and centered. If your favorite Web browser doesn't allow you to do that, download the notes and print them from a standalone PDF reader such as Adobe Acrobat.

Feedback on these notes is welcome. Please send comments to `alexis@clarkson.edu`.

# Chapter 1

# Introduction

In this chapter, we introduce the subject of these notes, automata theory and computability theory. We explain what this is and why it is worth studying.

Computer science can be divided into two main branches. One branch is concerned with the design and implementation of computer systems, with a special focus on software. (Computer hardware is the main focus of computer engineering.) This includes not only software development but also research into subjects like operating systems and computer security. This branch of computer science is called *applied computer science*. It can be viewed as the engineering component of computer science.

The other branch of computer science is the mathematical study of computation. One of its main goals is to determine which computational problems can and cannot be solved, and which ones can be solved efficiently. This involves discovering algorithms for computational problems, but also finding mathematical proofs that such algorithms do not exist. This branch of computer science is called *theoretical computer science* or the *theory of computation*.[1] It can be viewed

[1] The two main US-based theory conferences are the ACM's *Symposium on Theory of Comput-*

as the science component of computer science.[2]

To better illustrate what theoretical computer science is, consider the *Halting Problem*. The input to this computational problem is a program $P$ written in some fixed programming language. To keep things simple and concrete, let's limit the problem to C++ programs whose only input is a single text file. The output of the Halting Problem is the answer to the following question: Does the program $P$ halt on every possible input? In other words, does $P$ always halt no matter how it is used?

It should be clear that the Halting Problem is both natural and relevant. Software developers already have a tool that determines if the programs they write are correctly written according to the rules of the language they are using. This is part of what the compiler does. It would clearly be useful if software developers also had a tool that could determine if their programs are guaranteed to halt on every possible input.

Unfortunately, it turns out such a tool does not exist. Not that it currently does not exist and that maybe one day someone will invent one. No, it turns out that there is no algorithm that can solve the Halting Problem.

How can something like that be known? How can we know for sure that a computational problem has no algorithms — none, not now, not ever? Perhaps the only way of being absolutely sure of anything is to use mathematics. What this means is that we will define the Halting Problem precisely and then prove a theorem that says that no algorithm can solve the Halting Problem.

Note that this also requires that we define precisely what we mean by an algorithm or, more generally, what it means to compute something. Such a def-

*ing* (STOC) and the IEEE's *Symposium on Foundations of Computing* (FOCS). One of the main European theory conferences is the *Symposium on Theoretical Aspects of Computer Science* (STACS).

[2]Note that many areas of computer science have both applied and theoretical aspects. Artificial intelligence is one example. Some people working in AI produce actual systems that can be used in practice. Others try to discover better techniques by using theoretical models.

inition is called a *model of computation*. We want a model that's simple enough so we can prove theorems about it. But we also want this model to be close enough to real-life computation so we can claim that the theorems we prove say something that's relevant about real-life computation.

The general model of computation we will study is the *Turing machine*. We won't go into the details right now but the Turing machine is easy to define and, despite the fact that it is very simple, it captures the essential operations of real-life digital computers.[3]

Once we have defined Turing machines, we will be able to prove that no Turing machine can solve the Halting Problem. And we will take this as clear evidence that no real-life computer program can solve the Halting Problem.

The above discussion was meant to give you a better idea of what theoretical computer science is. The discussion focused on computability theory, the study of which computational problems can and cannot be solved. It's useful at this point to say a bit more about why theoretical computer science is worth studying (either as a student or as a researcher).

First, theoretical computer science provides critical guidance to applied computer scientists. For example, because the Halting Problem cannot be solved by any Turing machine, applied computer scientists do not waste their time trying to write programs that solves this problem, at least not in full generality. There are many other other examples, many of which are related to software verification.

Second, theoretical computer science involves concepts and techniques that have found important applications. For example, *regular expressions*, and algorithms that manipulate them, are used in compiler design and in the design of many programs that process input.

[3] In fact, the Turing machine was used by its inventor, Alan Turing, as a basic mathematical blueprint for the first digital computers.

Third, theoretical computer science is intellectually interesting, which leads some people to study it just out of curiosity. This should not be underestimated: many important scientific and mathematical discoveries have been made by people who were mainly trying to satisfy their intellectual curiosity. A famous example is number theory, which plays a key role in the design of modern cryptographic systems. The mathematicians who investigated number theory many years ago had no idea that their work would make possible electronic commerce as we know it today.

The plan for the rest of these notes is as follows. The first part of the notes will focus on *finite automata*, which are essentially Turing machines without memory. The study of finite automata is good practice for the study of general Turing machines. But we will also learn about the regular expressions we mentioned earlier. Regular expressions are essentially a way of describing patterns in strings. We will learn that regular expressions and finite automata are equivalent, in the sense that the patterns that can be described by regular expressions are also the patterns that can be recognized by finite automata. And we will learn algorithms that can convert regular expressions into finite automata, and vice-versa. These are the algorithms that are used in the design of programs, such as compilers, that process input. We will also learn how to prove that certain computational problems cannot be solved by finite automata, which also means that certain patterns cannot be described by regular expressions.

Next, we will study *context-free grammars*, which are essentially an extension of regular expressions that allows the description of more complex patterns. Context-free grammars are needed for the precise definition of modern programming languages and therefore play a critical role in the design of compilers.

The final part of these notes will focus on general Turing machines and will culminate in the proof that certain problems, such as the Halting Problem, cannot be solved by Turing machines.

# Chapter 2

# Finite Automata

In this chapter, we study a very simple model of computation called a finite automaton. Finite automata are useful for solving certain problems but studying finite automata is also good practice for the study of Turing machines, the general model of computation we will study later in these notes.

## 2.1 Turing Machines

As explained in the previous chapter, we need to define precisely what we mean by an algorithm, that is, what we mean when we say that something is computable. Such a definition is called a *model of computation*. As we said, we want a model that's simple enough so we can prove theorems about it, but a model that's also close enough to real-life computation so we can claim that the theorems we prove about our model say something that's relevant about real-life computation.

A first idea for a model of computation is any of the currently popular high-level programming languages. C++, for example. A C++ program consists

of instructions and variables. We could define C++ precisely but C++ is not simple. It has complex instructions such as loops and conditional statements, which can be nested within each other, as well as various other features such as type conversions, parameter passing and inheritance.

A simpler model would be a low-level assembler or machine language. These languages are much simpler. They have no variables, no functions, no types and only very simple instructions. An assembler program is a linear sequence of instructions (no nesting). These instructions directly access data that is stored in memory or in a small set of registers. One of these registers is the program counter, or instruction counter, that keeps track of which instruction is currently being executed. Typical instructions allow you to set the contents of a memory location to a given value, or copy the contents of a memory location to another one. Despite their simplicity, it is widely accepted that anything that can be computed by a C++ program can be computed by an assembler program. The evidence is that we have compilers that translate C++ programs into assembler.

But it is possible to define an even simpler model of computation. In this model, instructions can no longer access memory locations directly by specifying their address. Instead, we have a *memory head* that points to a single memory location. Instructions can access the data under the memory head and then move the head one location to the right or one location to the left. In addition, there is only one type of instruction in this model. Each instruction specifies, for each possible value that could be stored in the current memory location (the one under the memory head), a new value to be written at that location as well as the direction in which the memory head should be moved. For example, if $a, b, c$ are the possible values that could be stored in each memory location, the table in Figure 2.1 describes one possible instruction. Such a table is called a *transition table*. A program then consists of a simple loop that executes one of these instructions.

| | |
|---|---|
| $a$ | $b, R$ |
| $b$ | $b, L$ |
| $c$ | $a, R$ |

Figure 2.1: A transition table

| | $a$ | $b$ | $c$ |
|---|---|---|---|
| $q_0$ | $q_1, b, R$ | $q_0, a, L$ | $q_2, b, R$ |
| $q_1$ | $q_1, a, L$ | $q_1, c, R$ | $q_0, b, R$ |
| $q_2$ | $q_0, c, R$ | $q_2, b, R$ | $q_1, c, L$ |

Figure 2.2: A transition table

This is a very simple model of computation but we may have gone too far: since the value we write in each memory location depends only on the contents of that memory location, there is no way to copy a value from one memory location to another. To fix this without reintroducing more complicated instructions, we simply add to our model a special register we call the *state* of the program. Each instruction will now have to consider the current state in addition to the current memory location. A sample transition table is shown in Figure 2.2, assuming that $q_0, q_1, q_2$ are the possible states.

The model we have just described is called the *Turing machine*. (Each "program" in this model is considered a "machine".) To complete the description of the model, we need to specify a few more details. First, we restrict our atten-

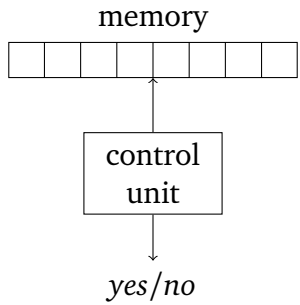



Figure 2.3: A Turing machine

tion to *decision problems*, which are computational problems in which the input is a string and the output is the answer to some yes/no question about the input. The Halting Problem mentioned in the previous chapter is an example of a decision problem.

Second, we need to describe how the input is given to a Turing machine, how the output is produced by the machine, and how the machine terminates its computation. For the input, we assume that initially, the memory of the Turing machine contains the input and nothing else. For the output, each Turing machine will have special *yes* and *no* states. Whenever one of these states is entered, the machine halts and produces the corresponding output. This takes care of termination too.

Figure 2.3 shows a Turing machine. The *control unit* consists of the transition table and the state.

The Turing machine is the standard model that is used to study computation mathematically. (As mentioned earlier, the Turing machine was used by its inventor, Alan Turing, as a basic mathematical blueprint for the first digital computers.) The Turing machine is clearly a simple model but it is also relevant

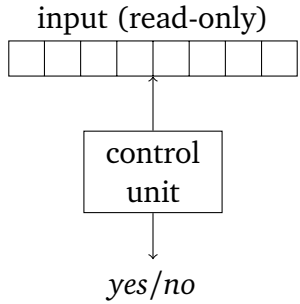



Figure 2.4: A finite automaton

to real-life computations because it is fairly easy to write a C++ program that can simulate any Turing machine, and because Turing machines are powerful enough to simulate typical assembler instructions. This last point will become clearer later in these notes when we take a closer look at Turing machines.

## 2.2 Introduction to Finite Automata

Before studying general Turing machines, we will study a restriction of Turing machines called *finite automata*. A finite automaton is essentially a Turing machine without memory. A finite automaton still has a state and still accesses its input one symbol at a time but the input can only be read, not written to, as illustrated in Figure 2.4. We will study finite automata for at least three reasons. First, this will help us understand what can be done with the control unit of a Turing machine. Second, this will allow us to learn how to study computation mathematically. Finally, finite automata are actually useful for solving certain types of problems.

| | underscore | letter | digit | other |
|---|---|---|---|---|
| $q_0$ | $q_1$ | $q_1$ | $q_2$ | $q_2$ |
| $q_1$ | $q_1$ | $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_2$ | $q_2$ | $q_2$ | $q_2$ |

Figure 2.5: The transition table of a finite automaton that determines if the input is a valid C++ identifiers

Here are a few more details about finite automata. First, in addition to being read-only, the input must be read from left to right. In other words, the input head can only move to the right. Second, the computation of a finite automaton starts at the beginning of the input and automatically ends when the end of the input is reached, that is, after the last symbol of the input has been read. Finally, some of the states of the automaton are designated as *accepting states*. If the automaton ends its computation in one of those states, then we say that the input is *accepted*. In other words, the output is *yes*. On the other hand, if the automaton ends its computation in a non-accepting state, then the input is *rejected* — the output is *no*.

Now that we have a pretty good idea of what a finite automaton is, let's look at a couple of examples. A valid C++ identifier consists of an underscore or a letter followed by any number of underscores, letters and digits. Consider the problem of determining if an input string is a valid C++ identifier. This is a decision problem.

Figure 2.5 shows the transition table of a finite automaton that solves this problem. As implied by the table, the automaton has three states. State $q_0$ is the *start state* of the automaton. From that state, the automaton enters state $q_1$ if it the first symbol of the input is an underscore or a letter. The automaton will

remain in state $q_1$ as long as the input consists of underscores, letter and digits. In other words, the automaton finds itself in state $q_1$ if the portion of the string it has read so far is a valid C++ identifier.

On the other hand, the automaton enters state $q_2$ if it has decided that the input cannot be a valid C++ identifier. If the automaton enters state $q_2$, it will never be able to leave. This corresponds to the fact that once we see a character that causes the string to be an invalid C++ identifier, there is nothing we can see later that could fix that. A non-accepting state from which you cannot escape is sometimes called a *garbage state*.

State $q_1$ is accepting because if the computation ends in that state, then this implies that the entire string is a valid C++ identifier. The start state is not an accepting state because the empty string, the one with no characters, is not a valid identifier.

Let's look at some sample strings. When reading the string `input_file`, the automaton enters state $q_1$ and remains there until the end of the string. Therefore, this string is accepted, which is correct. On the other hand, when reading the string `input-file`, the automaton enters state $q_1$ when it sees the first `i` but then leaves for state $q_2$ when it encounters the dash. This will cause the string to be rejected, which is correct since dashes are not allowed in C++ identifiers.

A finite automaton can also be described by using a graph. Figure 2.6 shows the *transition graph* of the automaton for valid C++ identifiers. Each state is represented by a node in this graph. Each edge connects two states and is labeled by an input symbol. If an edge goes from $q$ to $q'$ and is labeled $a$, this indicates that when in state $q$ and reading an $a$, the automaton enters state $q'$. Each such step is called a *move* or a *transition* (hence the terms transition table and transition graph). The start state is indicated with an arrow and the accepting states have a double circle.

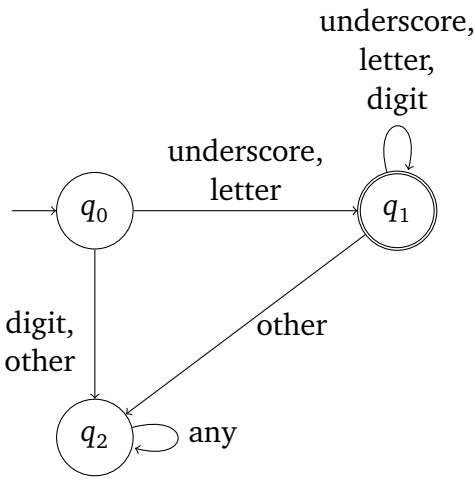



Figure 2.6: The transition graph of a finite automaton that determines if the input is a valid C++ identifier

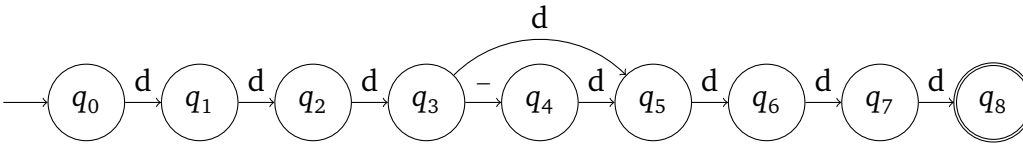



Figure 2.7: A finite automaton that determines if the input is a correctly formatted phone number

Transition graphs and transition tables provide exactly the same information. But the graphs make it easier to visualize the computation of the automaton, which is why we will draw transition graphs whenever possible. There are some circumstances, however, where it is impractical to draw a graph. We will see examples later.

Let's consider another decision problem, the problem of determining if an input string is a phone number in one of the following two formats: 7 digits, or 3 digits followed by a dash and 4 digits. Figure 2.7 shows a finite automaton that solves this problem. The transition label `d` stands for `digit`.

In a finite automaton, from every state, there should be a transition for every possible input symbol. This means that the graph of Figure 2.7 is missing many transitions. For example, from state $q_0$, there is no transition labeled – and there is no transition labeled by a letter. All those missing transitions correspond to cases in which the input string should be rejected. Therefore, we can have all of those missing transitions go to a garbage state. We chose not to draw those transitions and the garbage state simply to avoid cluttering the diagram unnecessarily.

It is interesting to compare the finite automata we designed in this section with C++ programs that solve the same problems. For example, Figure 2.8 shows an algorithm that solves the C++ identifier problem. The algorithm is

```
if (end of input)
    return no
read char c
if (c is not underscore or letter)
    return no
while (not end of input)
    read char c
    if (c is not underscore, letter or digit)
        return no
return yes
```

Figure 2.8: A simple algorithm that determines if the input is a valid C++ identifier

described in pseudocode, which is simpler than C++. But even by looking at pseudocode, the simplicity of finite automata is evident. The pseudocode algorithm uses variables and different types of instructions. The automaton, on the other hand, consists of only states and transitions. This is consistent with what we said earlier, that Turing machines, and therefore finite automata, are a much simpler model of computation that a typical high-level programming language.

At the beginning of this section, we said that finite automata can be useful for solving certain problems. This relies on the fact that finite automata can be easily converted into programs, as shown in Figure 2.9. All that we need to add to this algorithm is a function `start_state()` that returns the start state of the finite automaton, a function `next_state(q, c)` that, for every pair `(q, c)` where `q` is a state and `c` is an input character, returns the next state according to the transition table (or graph) of the finite automaton, and a function `is_accepting(q)` that returns `true` if state `q` is an accepting state.

Therefore, a possible strategy for solving a decision problem is to design a

```
q = start_state()
while (not end of input)
    read symbol c
    q = next_state(q, c)
if (is_accepting(q))
    return yes
else
    return no
```

Figure 2.9: An algorithm that simulates a finite automaton

finite automaton and then convert it to a pseudocode algorithm as explained above. In some cases, this can be easier than designing a pseudocode algorithm directly, for several reasons. One is that the simplicity of the finite automaton model can help us focus more clearly on the problem itself, without being distracted by the various features that can be used in a high-level pseudocode algorithm. Another reason is that the computation of a finite automaton can be visualized by its transition graph, making it easier to understand what is going on. Finally, when designing a finite automaton we have to include transitions for every state and every input symbol. This helps to ensure that we consider all the necessary cases. This strategy of designing finite automata and then converting them to algorithms is used in the design of compilers and other programs that perform input processing.

## Study Questions

2.2.1. Can finite automata solve problems that pseudocode algorithms cannot?

2.2.2. What are three advantages of finite automata over pseudocode algo-

rithms?

## Exercises

2.2.3. Consider the problem of determining if a string is an integer in the following format: an optional minus sign followed by at least one digit. Design a finite automaton for this problem.

2.2.4. Consider the problem of determining if a string is a number in the following format: an optional minus sign followed by at least one digit, or an optional minus sign followed by any number of digits, a decimal point and at least one digit. Design a finite automaton for this problem.

2.2.5. Suppose that a valid C++ identifier is no longer allowed to consist of only underscores. Modify the finite automaton of Figure 2.6 accordingly.

2.2.6. Add optional area codes to the phone number problem we saw in this section. That is, consider the problem of determining if a string is a phone number in the following format: 7 digits, or 10 digits, or 3 digits followed by a dash and 4 digits, or 3 digits followed by a dash, 3 digits, another dash and 4 digits. Design a finite automaton for this problem.

2.2.7. Convert the finite automaton of Figure 2.6 into a high-level pseudocode algorithm by using the technique explained in this section. That is, write pseudocode for the functions `starting_state()`, `next_state(q, c)` and `is_accepting(q)` of Figure 2.9.

2.2.8. Repeat the previous exercise for the finite automaton of Figure 2.7. (Don't forget the garbage state.)

## 2.3 More Examples

The examples of finite automata we have seen so far, in the text and in the exercises, have been inspired by real-world applications. We now take a step back and consider what else finite automata can do. We will consider several problems that may not have obvious applications but that illustrate basic techniques that are useful in the design of finite automata.

But first, we define precisely — that is, mathematically — several useful concepts. The main purpose of these definitions is to define terms that allow us to more conveniently talk about finite automata and the problems they solve.

**Definition 2.1** *An* alphabet *is a finite set whose elements are called* symbols*.*

This definition allows us to talk about the *input alphabet* of a finite automaton instead of having to say the *set of possible input symbols* of a finite automaton. In this context, symbols are sometimes also called letters or characters, as we did in the previous section.

**Definition 2.2** *A* string *over an alphabet A is a finite sequence of symbols from A.*

The *length* of a string is the number of symbols it contains. Note that a string can be empty: the *empty string* has length 0 and contains no symbols. In these notes, we will use $\varepsilon$ to denote the empty string.

Now, each finite automaton solves a problem by accepting some strings and rejecting the others. Another way of looking at this is to say that the finite automaton *recognizes* a set of strings, those that it accepts.

**Definition 2.3** *A* language *over an alphabet A is a set of strings over A.*

**Definition 2.4** *The language* recognized *by a finite automaton M (or the language* of *M) is the set of strings accepted by M.*

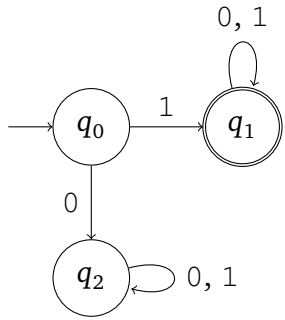



Figure 2.10: A DFA for the language of strings that start with 1

Note that, strictly speaking, this definition is not complete because we haven't defined precisely — mathematically — what a finite automaton is and what it means for a finite automaton to accept a string. We will do that in the next section. Our informal understanding of finite automata and how they work will suffice for now.

Also note that the type of finite automaton we have been discussing is usually referred to as a *deterministic* finite automaton, usually abbreviated *DFA*. We will enounter nondeterministic finite automata (NFA's) later in these notes.

We are now ready to look at more examples of DFA's. Unless otherwise specified, in the examples of this section, the input alphabet is $\{0, 1\}$.

**Example 2.5** Figure 2.10 shows a DFA for the language of strings that start with 1. □

**Example 2.6** Now consider the language of strings that *end* in 1. One difficulty here is that there is no mechanism in a DFA that allows us to know whether the symbol we are currently looking at is the last symbol of the input string. So we

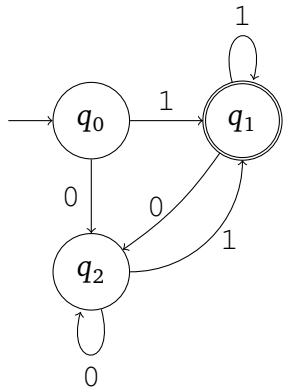



Figure 2.11: A DFA for the language of strings that end in `1`

have to always be ready, as if the current symbol was the last one.[1] What this means is that after reading every symbol, we have to be in an accept state if and only if the portion of the input string we've seen so far is in the language.

Figure 2.11 shows a DFA for this language. Strings that begin in `1` lead to state $q_1$ while strings that begin in `0` lead to state $q_2$. Further `0`'s and `1` cause the DFA to move between these states as needed.

Notice that the starting state is not an accepting state because the empty string, the string of length 0 that contains no symbols, does not end in `1`. But then, states $q_0$ and $q_2$ play the same role in the DFA: they're both non-accepting states and the transitions coming out of them lead to the same states. This implies that these states can be merged to get the slightly simpler DFA shown in Figure 2.12. □

[1] To paraphrase a well-known quote attributed to Jeremy Schwartz, "Read every symbol as if it were your last. Because one day, it will be."

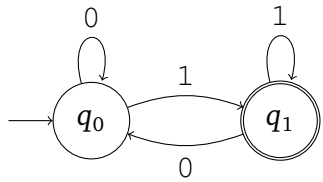



Figure 2.12: A simpler DFA for the language of strings that end in 1

**Example 2.7** Consider the language of strings of length at least two that begin and end with the same symbol. A DFA for this language can be obtained by combining the ideas of the previous two examples, as shown in Figure 2.13 □

**Example 2.8** Consider the language of strings that contain the string 001 as a substring. What this means is that the symbols 0, 0, 1 occur *consecutively* within the input string. For example, the string 0100110 is in the language but 0110110 is not.

Figure 2.14 shows a DFA for this language. The idea is that the DFA remembers the longest *prefix* of 001 that ends the portion of the input string that has been seen so far. For example, initially, the DFA has seen nothing, so the starting state corresponds to the empty string. If the DFA then sees a 0, it moves to state $q_1$. If it then sees a 1, then the portion of the input string that the DFA has seen so far ends in 01, which is not a prefix of 001. So the DFA goes back to state $q_0$. □

**Example 2.9** Consider the language of strings that contain an even number of 1's. Initially, the number of 1's is 0, which is even. After seeing the first 1, that number will be 1, which is odd. After seeing each additional 1, the DFA will toggle back and forth between even and odd. This idea leads to the DFA shown in Figure 2.15. Note how the input symbol 0 never affects the state of the DFA.

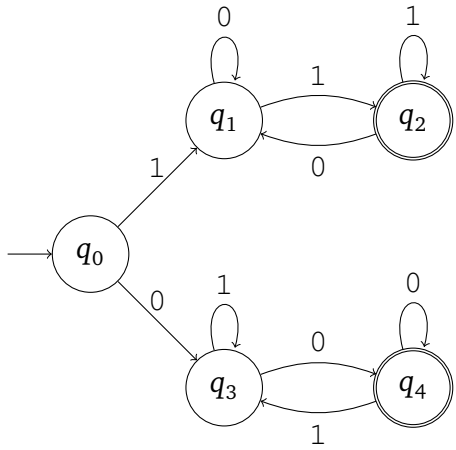



Figure 2.13: A DFA for the language of strings of length at least two that begin and end with the same symbol

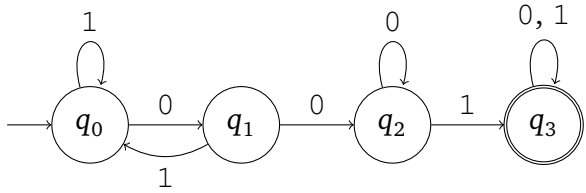



Figure 2.14: A DFA for the language of strings that contain the substring `001`

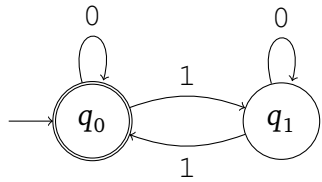



Figure 2.15: A DFA for the language of strings that contain an even number of 1's

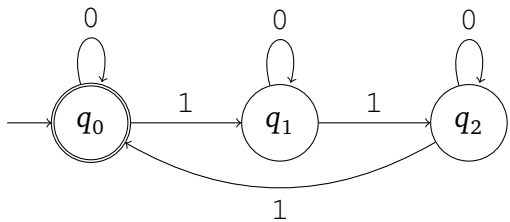



Figure 2.16: A DFA for the language of strings that contain a number of 1's that's a multiple of 3

We say that in this DFA, and with respect to this language, the symbol is *neutral*. □

**Example 2.10** The above example can be generalized. A number is even if it is a multiple of 2. So consider the language of strings that contain a number of 1's that's a multiple of 3. The idea is to count modulo 3, as shown in Figure 2.16. □

**Example 2.11** We can generalize this even further. For every number $k \geq 2$, consider the language of strings that contain a number of 1's that's a multiple of

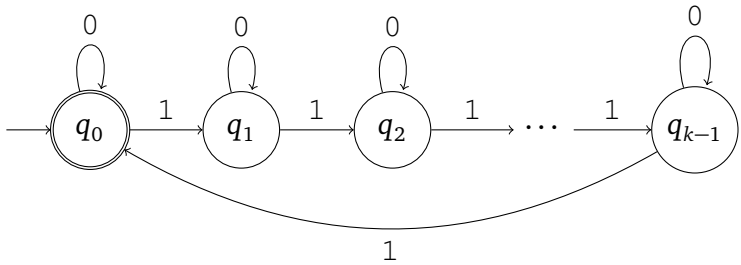



Figure 2.17: A DFA for the language of strings that contain a number of 1's that's a multiple of $k$

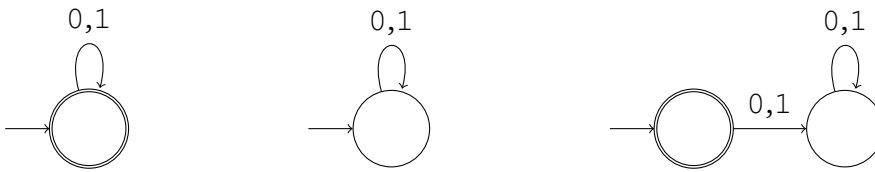



Figure 2.18: DFA's for the languages $\Sigma^*$, $\emptyset$ and $\{\varepsilon\}$

$k$. Note that this defines an infinite number of languages, one for every possible value of $k$. Each one of those languages can be recognized by a DFA since, for every $k \geq 2$, a DFA can be constructed to count modulo $k$, as shown in Figure 2.17. □

**Example 2.12** We end this section with three simple DFA's for the basic but important languages $\Sigma^*$ (the language of all strings), $\emptyset$ (the empty language) and $\{\varepsilon\}$ (the language that contains only the empty string). The DFA's are shown in Figure 2.18. □

## Study Questions

2.3.1. What is an alphabet?

2.3.2. What is a string?

2.3.3. What is a language?

2.3.4. What does it mean for a language to be recognized by a DFA?

## Exercises

2.3.5. Modify the DFA of Figure 2.13 so that strings of length 1 are also accepted.

2.3.6. Give DFA's for the following languages. In all cases, the alphabet is $\{0, 1\}$.

a) The language of strings of length at least two that begin with `0` and end in `1`.

b) The language of strings of length at least two that have a `1` as their second symbol.

c) The language of strings of length at least $k$ that have a `1` in position $k$. Do this in general, for every $k \geq 1$. (You did the case $k = 2$ in part (b).)

2.3.7. Give DFA's for the following languages. In all cases, the alphabet is $\{0, 1\}$.

a) The language of strings that contain at least one `1`.

b) The language of strings that contain exactly one `1`.

c) The language of strings that contain at least two `1`'s.

d) The language of strings that contain less than two `1`'s.

e) The language of strings that contain at least $k$ `1`'s. Do this in general, for every $k \geq 0$. (You did the case $k = 2$ in part (c).)

2.3.8. Give DFA's for the following languages. In all cases, the alphabet is $\{0, 1\}$.

a) The language of strings of length at least two whose first two symbols are the same.

b) The language of strings of length at least two whose last two symbols are the same.

c) The language of strings of length at least two that have a `1` in the second-to-last position.

## 2.4 Formal Definition

In this section, we define precisely what we mean by a DFA. Once again, what we are looking for is a mathematically precise definition. We will call this a *formal* definition. Such a definition is necessary for proving mathematical statements about DFA's and for writing programs that manipulate DFA's.

From the previous sections, it should be clear that a DFA consists of four things:

- A finite set of states.
- A special state called the starting state.
- A subset of states called accepting states.
- A transition table or graph that specifies a next state for every possible pair (state, input character).

Actually, to be able to specify all the transitions, we also need to know what the possible input symbols are. This information should also be considered part of the DFA:

- A set of possible input symbols.

We can also define what it means for a DFA to accept a string: run the algorithm of Figure 2.9 and accept if the algorithm returns yes.

The above definition of a DFA and its operation should be pretty clear. But it has a couple of problems. The first one is that it doesn't say exactly what a transition table or graph is. That wouldn't be too hard to fix but the second problem with the above definition is more serious: the operation of the DFA is defined in terms of a pseudocode algorithm. For this definition to be complete, we would need to also define what those algorithms are. But recall that we are interested in DFA's mainly because they are supposed to be easy to define. DFA's are not going to be simpler than pseudocode algorithms if the definition of a DFA includes a pseudocode algorithm.

In the rest of this section, we will see that it is possible to define a DFA and its operation without referring to either graphs, tables or algorithms. This will be our *formal* definition. The above definition, in terms of a graph or table, and an algorithm, will be considered an informal definition.

**Definition 2.13** *A* deterministic finite automaton (DFA) *is a* 5*-tuple* $(Q, \Sigma, \delta, q_0, F)$ *where*

1. $Q$ *is a finite set of* states.[2]

2. $\Sigma$ *is an alphabet called the* input alphabet.

[2]It would be more correct to say that $Q$ is a finite set whose elements are called *states*, as we did in the definition of an alphabet. But this shorter style is easier to read.

3. $\delta : Q \times \Sigma \to Q$ *is the* transition function.

4. $q_0 \in Q$ *is the* starting state.

5. $F \subseteq Q$ *is the set of* accepting states.

As mentioned in the previous section, this definition is for deterministic finite automata. We will encounter nondeterministic finite automata later in these notes.

When we describe a DFA, we usually draw a transition graph because it helps us visualize the computation of the automaton. But there are situations where transition graphs (or tables) are not practical. It could be because the automaton has too many states or because the transitions are hard to understand. We will see an example later in this section.

In those situations, we can describe the DFA as a 5-tuple, as in the formal definition. We call this a *formal description* of the DFA. In a formal description, the transition function could, in principle, be described by a diagram similar to a transition graph, or by a table. But if we are giving a formal description of a DFA, it's usually because a graph or table is not practical. This is why, in a formal description, the transition function is normally described by a set of equations. Here's an example.

**Example 2.14** Consider the DFA of Figure 4.1. Here's a formal description of this DFA. The DFA is $(\{q_0, q_1, q_2\}, \Sigma, \delta, q_0, \{q_1\})$ where $\Sigma$ is the set of all characters that appear on a standard keyboard.

Before describing the transition function $\delta$, it is useful to explain the role that each state plays in the DFA. State $q_0$ is just a start state. State $q_1$ is entered if the input is a valid identifier. State $q_2$ is a garbage state.

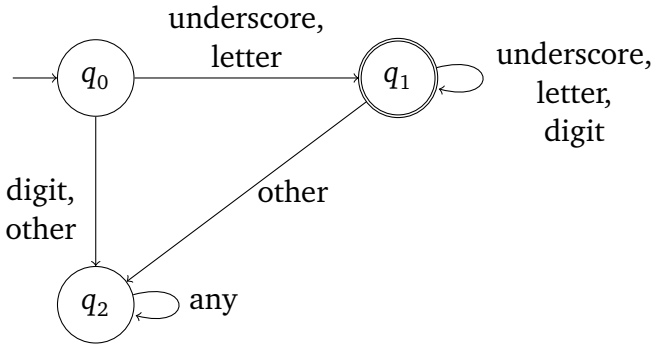



Figure 2.19: The transition graph of a DFA for C++ identifiers

The transition function of the DFA can then be described as follows:

$$\delta(q_0, c) = \begin{cases} q_1 & \text{if } c \text{ is an underscore or a letter} \\ q_2 & \text{otherwise} \end{cases}$$

$$\delta(q_1, c) = \begin{cases} q_1 & \text{if } c \text{ is an underscore, a letter or a digit} \\ q_2 & \text{otherwise} \end{cases}$$

$$\delta(q_2, c) = q_2, \quad \text{for every } c \in \Sigma.$$

□

We now define what it means for a DFA to accept its input string. Without referring to an algorithm. Instead, we will only talk about the sequence of states that the DFA goes through while processing its input string.

**Definition 2.15** *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w = w_1 \cdots w_n$ be a*

*string of length n over* $\Sigma$.[3] *Let* $r_0, r_1, \ldots, r_n$ *be the sequence of states defined by*

$$r_0 = q_0$$
$$r_i = \delta(r_{i-1}, w_i), \quad \textit{for } i = 1, \ldots, n.$$

*Then* $M$ accepts $w$ *if and only if* $r_n \in F$.

Note how the sequence of states is defined by only referring to the transition function, without referring to an algorithm that *computes* that sequence of states.

In the previous section, we defined the language of a DFA as the set of strings accepted by the DFA. Now that we have formally defined what a DFA is and what it means for a DFA to accept a string, that definition is complete. Here it is again, for completeness:

**Definition 2.16** *The language* recognized *by a DFA* $M$ *(or the language* of $M$*) is the following set:*

$$L(M) = \{w \in \Sigma^* \mid w \textit{ is accepted by } M\}$$

*where* $\Sigma$ *is the input alphabet of* $M$ *and* $\Sigma^*$ *denotes the set of all strings over* $\Sigma$.

The languages that are recognized by DFA's are called *regular*.

**Definition 2.17** *A language is* regular *if it is recognized by some DFA.*

Many interesting languages are regular but we will soon learn that many others are not. Those languages require algorithms that are more powerful than DFA's.

[3] It is understood here that $w_1, \ldots, w_n$ are the individual symbols of $w$.

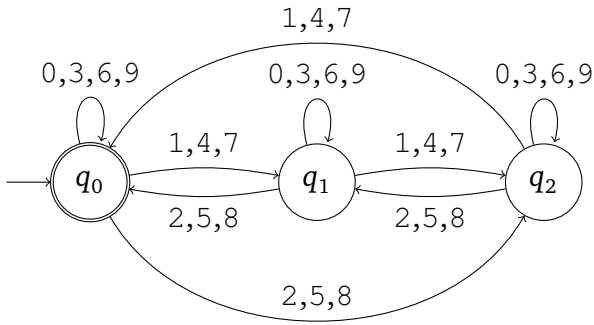



Figure 2.20: A DFA for the language of strings whose digits add to a multiple of 3

We end this section with two additional examples of DFA's. The second one will generalize the first one and, in the process, demonstrate the usefulness of formal descriptions.

**Example 2.18** Let's go back the modulo 3 counting example and generalize it in another way. Suppose that the input alphabet is now $\{0, 1, 2, \ldots, 9\}$ and consider the language of strings that have the property that the sum of their digits is a multiple of 3. For example, 315 is in the language because $3+1+5 = 9$ is a multiple of 3. The idea is still to count modulo 3 but there are now more cases to consider, as shown in Figure 2.20. □

**Example 2.19** Now let's go all out and combine the generalizations of the last two examples. That is, over the alphabet $\{0, 1, 2, \ldots, 9\}$, for every number $k$, let's consider the language of strings that have the property that the sum of their digits is a multiple of $k$. Once again, the idea is to count modulo $k$. For example, Figure 2.21 shows the DFA for $k = 4$.

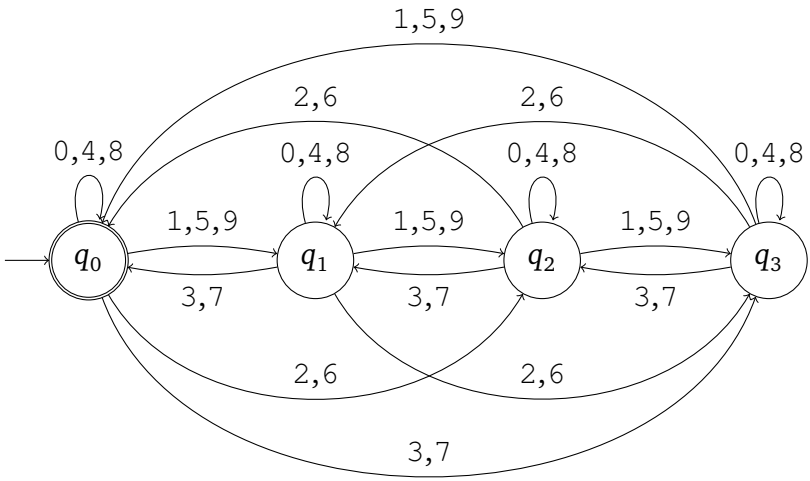



Figure 2.21: A DFA for the language of strings whose digits add to a multiple of 4

It should be pretty clear that a DFA exists for every $k$. But the transition diagram would be difficult to draw and it would likely be ambiguous. This is an example where we are better off describing the DFA by giving a formal description. Here it is: the DFA is $(Q, \Sigma, \delta, q_0, F)$ where

$$Q = \{q_0, q_1, q_2, \ldots, q_{k-1}\}$$
$$\Sigma = \{0, 1, 2, \ldots, 9\}$$
$$F = \{q_0\}$$

and $\delta$ is defined as follows: for every $i \in Q$ and $c \in \Sigma$,

$$\delta(q_i, c) = q_j, \quad \text{where } j = (i + c) \bmod k.$$

□

## Study Questions

2.4.1. What is the advantage of the formal definition of a DFA over the informal definition presented at the beginning of this section?

2.4.2. What is a DFA? (Give a formal definition.)

2.4.3. What does it mean for a DFA to accept a string? (Give a formal definition.)

2.4.4. What is a regular language?

## Exercises

2.4.5. Give formal descriptions of the following DFA's. In each case, describe the transition function using equations. (Note that the DFA's of this exercise

are better described by transition graphs than by formal descriptions. This is just an exercise.)

a) The DFA of Figure 2.7. (Don't forget the garbage state.)

b) The DFA of Exercise 2.2.3.

c) The DFA of Exercise 2.2.4.

2.4.6. Give a DFA for the language of strings of length at least $k$ that have a `1` in position $k$ from the end. Do this in general, for every $k \geq 1$. The alphabet is $\{0, 1\}$. (You did the case $k = 2$ in Exercise 2.3.8, part (c).)

2.4.7. Give DFA's for the following languages. In both cases, the alphabet is the set of digits $\{0, 1, 2, \ldots, 9\}$.

a) The language of strings that represent a multiple of 3. For example, the string `036` is in the language because 36 is a multiple of 3.

b) The generalization of the previous language where 3 is replaced by any number $k \geq 2$.

2.4.8. This exercise asks you to show that DFA's can add, at least when the numbers are presented in certain ways. Consider the alphabet that consists of symbols of the form `[`$abc$`]` where $a$, $b$ and $c$ are digits. For example, `[631]` and `[937]` are two of the symbols in this alphabet. If $w$ is a string of digits, let $n(w)$ denote the number represented by $w$. For example, if $w$ is the string `428`, then $n(w)$ is the number 428. Now give DFA's for the following languages.

a) The language of strings of the form `[`$x_0y_0z_0$`]` `[`$x_1y_1z_1$`]` $\cdots$ `[`$x_ny_nz_n$`]`

such that

$$n(x_n \cdots x_1 x_0) + n(y_n \cdots y_1 y_0) = n(z_n \cdots z_1 z_0).$$

For example, `[279][864][102]` is in the language because $182 + 67 = 249$. (This language corresponds to reading the numbers from right to left and position by position. Note that this is how we "read" numbers when we add them by hand.)

b) The language of strings of the form $[x_n y_n z_n] \cdots [x_1 y_1 z_1] [x_0 y_0 z_0]$ such that

$$n(x_n \cdots x_1 x_0) + n(y_n \cdots y_1 y_0) = n(z_n \cdots z_1 z_0).$$

For example, `[102][864][279]` is in the language because $182 + 67 = 249$. (This time, the numbers are read from left to right.)

## 2.5 Closure Properties

In the previous section, we considered the language of strings that contain the substring `001` and designed a DFA that recognizes it. The DFA is shown again in Figure 2.22. Suppose that we are now interested in the language of strings that do *not* contain the substring `001`. Is that language regular?

A quick glance at the DFA should reveal a solution: since strings that contain `001` end up in state $q_3$ and strings that do not contain `001` end up in one of the other states, all we have to do is switch the acceptance status of every state in the above DFA to obtain a DFA for this new language. So the answer is, yes, the new language is regular.

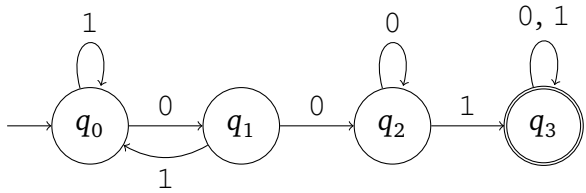



Figure 2.22: A DFA for the language of strings that contain the substring `001`

This new language is the *complement* of the first one.[4] And the above technique should work for any regular language $L$: by switching the accepting states of a DFA for $L$, we should get a DFA for $\overline{L}$.

In a moment, we will show in detail that the complement of a regular language is always regular. This is an example of what is called a *closure property*.

**Definition 2.20** *A set is* closed *under an operation if applying that operation to elements of the set results in an element of the set.*

For example, the set of natural numbers is closed under addition and multiplication but not under subtraction or division because $2-3$ is negative and $1/2$ is not even an integer. The set of integers is closed under addition, subtraction and multiplication but not under division. The set of rational numbers is closed under those four operations but not under the square root operation since $\sqrt{2}$ is not a rational number.[5]

[4]Note that to define precisely what is meant by the complement of a language, we need to know the alphabet over which the language is defined. If $L$ is a language over $\Sigma$, then its complement is defined as follows: $\overline{L} = \{w \in \Sigma^* \mid w \notin L\}$. In other words, $\overline{L} = \Sigma^* - L$.

[5]The proof of this fact is a nice example of a proof by contradiction and of the usefulness of basic number theory. Suppose that $\sqrt{2}$ is a rational number. Let $a$ and $b$ be positive integers such that $\sqrt{2} = a/b$. Since fractions can be simplified, we can assume that $a$ and $b$ have no

**Theorem 2.21** *The class of regular languages is closed under complementation.*

**Proof** Suppose that $A$ is regular and let $M$ be a DFA for $A$. We construct a DFA $M'$ for $\overline{A}$.

Let $M'$ be the result of switching the accepting status of every state in $M$. More precisely, if $M = (Q, \Sigma, \delta, q_0, F)$, then $M' = (Q, \Sigma, \delta, q_0, Q - F)$. We claim that $L(M') = \overline{A}$.

To prove that, suppose that $w \in A$. Then, in $M$, $w$ leads to an accepting state. This implies that in $M'$, $w$ leads to the same state but this state is non-accepting in $M'$. Therefore, $M'$ rejects $w$.

A similar argument shows that if $w \notin A$, then $M'$ accepts $w$. Therefore, $L(M') = \overline{A}$. □

Note that the proof of this closure property is *constructive*: it establishes the existence of a DFA for $\overline{A}$ by providing an algorithm that constructs that DFA. Proofs of existence are not always constructive. But when they are, the algorithms they provide are often useful.

At this point, it is natural to wonder if the class of regular languages is closed under other operations. A natural candidate is the *union* operation: if $A$ and $B$ are two languages over $\Sigma$, then the union of $A$ and $B$ is $A \cup B = \{w \in \Sigma^* \mid w \in A \text{ or } w \in B\}$.[6]

---

common factors (other than 1). Now, the fact that $\sqrt{2} = a/b$ implies that $2 = a^2/b^2$ and that $2b^2 = a^2$. Therefore, $a$ is even and 4 divides $a^2$. But then, 4 must also divide $2b^2$, which implies that 2 divides $b$. Therefore, 2 divides both $a$ and $b$, contradicting the fact that $a$ and $b$ have no common factors.

[6] We could also consider the union of languages that are defined over different alphabets. In that case, the alphabet for the union would be the union of the two underlying alphabets: if $A$ is a language over $\Sigma_1$ and $B$ is a language over $\Sigma_2$, then $A \cup B = \{w \in (\Sigma_1 \cup \Sigma_2)^* \mid w \in A \text{ or } w \in B\}$. In these notes, we will normally consider only the union of languages over the same alphabet because this keeps things simpler and because this is probably the most common situation that occurs in practice.

For example, consider the language of strings that contain either a number of 0's that's even or a number of 1's that's even. Call this language $L$. It turns out that $L$ is the union of two languages we are familiar with:

$$L = L_1 \cup L_2$$

where

$$L_1 = \{w \in \{0,1\}^* \mid w \text{ contains an even number of } 0\text{'s}\}$$

and

$$L_2 = \{w \in \{0,1\}^* \mid w \text{ contains an even number of } 1\text{'s}\}$$

In particular, we already know how to construct DFA's for $L_1$ and $L_2$. These DFA's are the first two shown in Figure 2.23.

Now, a DFA for $L$ can be designed by essentially simulating the DFA's for $L_1$ and $L_2$ *in parallel*. That is, let $M_1$ and $M_2$ be the DFA's for $L_1$ and $L_2$. The DFA for $L$, which we will call $M$, will "store" the current state of both $M_1$ and $M_2$, and update these states according to the transition functions of these DFA's. This can be implemented by having each state of $M$ be a pair that combines a state of $M_1$ with a state of $M_2$. The result is the third DFA shown in Figure 2.23.

The 0 transitions in $M$ toggle the first state in each pair between $q_0$ and $q_1$ (and between columns) while having no effect on the second state (and the row). This is consistent with the fact that 0 transitions toggle the state in $M_1$ and have no effect in $M_2$. The 1 transitions in $M$ toggle the second state in each pair between $r_0$ and $r_1$ (and between rows) while having no effect on the first state (and the column). This is consistent with the fact that 1 transitions toggle the state in $M_2$ and have no effect in $M_1$.

The accepting states of $M$ are those pairs that have $q_0$ has their first or $r_0$ as their second state. This is correct since $M$ should accept the input whenever either $M_1$ or $M_2$ accepts.

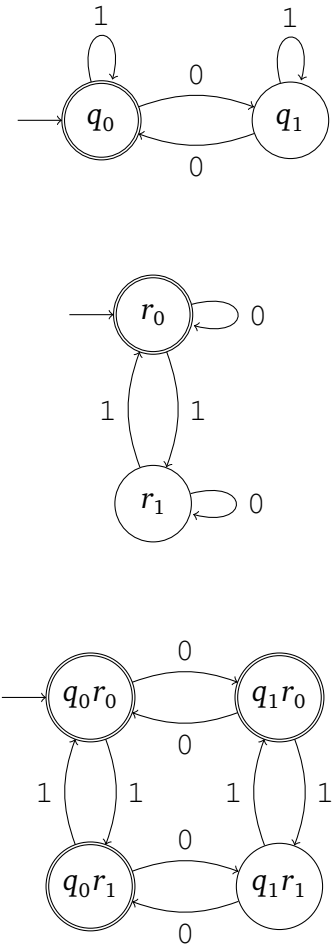



Figure 2.23: A DFA for the language of strings that contain an even number of 0's, a DFA for the language of strings that contain an even number of 1's, and a DFA for the union of these two languages

The above idea can be generalized to show that the union of any two regular languages is always regular.

**Theorem 2.22** *The class of regular languages is closed under union.*

**Proof** Suppose that $A_1$ and $A_2$ are regular and let $M_1$ and $M_2$ be DFA's for these languages. We construct a DFA $M$ that recognizes $A_1 \cup A_2$.

The idea, as explained above, is that $M$ is going to simulate $M_1$ and $M_2$ in parallel. More precisely, if after reading a string $w$, $M_1$ would be in state $r_1$ and $M_2$ would be in state $r_2$, then $M$, after reading the string $w$, will be in state $(r_1, r_2)$.

Here are the full details. Suppose that $M_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$, for $i = 1, 2$. Then let $M = (Q, \Sigma, \delta, q_0, F)$ where

$$\begin{aligned} Q &= Q_1 \times Q_2 \\ q_0 &= (q_1, q_2) \\ F &= \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\} \end{aligned}$$

and $\delta$ is defined as follows: for every $r_1 \in Q_1$, $r_2 \in Q_2$ and $a \in \Sigma$,

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

Because the start state of $M$ consists of the start states of $M_1$ and $M_2$, and because $M$ updates its state according to the transition functions of $M_1$ and $M_2$, it should be clear that after reading a string $w$, $M$ will be in state $(r_1, r_2)$ where $r_1$ and $r_2$ are the states that $M_1$ and $M_2$ would be in after reading $w$.[7] Now, if

[7] We could provide more details here, if we thought our readers needed them. The idea is to refer to the details in the definition of acceptance. Suppose that $w = w_1 \cdots w_n$ is a string of length $n$ and that $r_0, r_1, \ldots, r_n$ is the sequence of states that $M_1$ goes through while reading $w$.

$w \in A_1 \cup A_2$, then either $r_1 \in F_1$ or $r_2 \in F_2$, which implies that $M$ accepts $w$. The reverse is also true. Therefore, $L(M) = A_1 \cup A_2$. □

**Corollary 2.23** *The class of regular languages is closed under intersection.*

**Proof** In the proof of the previous theorem., change the definition of $F$ as follows:

$$F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ and } r_2 \in F_2\}.$$

In other words, $F = F_1 \times F_2$. □

So we now know that the class of regular languages is closed under the three basic set operations: complementation, union and intersection. And the proofs of these closure properties are all constructive.

We end this section with another operation that's specific to languages. If $A$ and $B$ are two languages over $\Sigma$, then the *concatenation* of $A$ and $B$ is

$$AB = \{xy \in \Sigma^* \mid x \in A \text{ and } y \in B\}.$$

---

This means that $r_0 = q_1$ and that, for $i = 1, \ldots, n$,

$$r_i = \delta(r_{i-1}, w_i).$$

Suppose that $s_0, s_1, \ldots, s_n$ is the sequence of states that $M_2$ goes through while reading $w$. Again, this means that $s_0 = q_2$ and that, for $i = 1, \ldots, n$,

$$s_i = \delta(s_{i-1}, w_i).$$

Then $(r_0, s_0), (r_1, s_1), \ldots, (r_n, s_n)$ has to be the sequence of states that $M$ goes through while reading $w$ because $(r_0, s_0) = (q_1, q_2) = q_0$ and, for $i = 1, \ldots, n$,

$$(r_i, s_i) = (\delta_1(r_{i-1}, w_i), \delta_2(s_{i-1}, w_i)) = \delta((r_{i-1}, s_{i-1}), w_i).$$

Therefore, if after reading $w$, $M_1$ is in state $r_n$ and $M_2$ is in state $s_n$, then $M$ is in state $(r_n, s_n)$.

That is, the concatenation of $A$ and $B$ consists of those strings we can form, in all possible ways, by taking a string from $A$ and appending to it a string from $B$.

For example, suppose that $A$ is the language of strings that consist of an even number of `0`'s (no `1`'s) and that $B$ is the language of strings that consist of an odd number of `1`'s (no `0`'s). That is,

$$A = \{0^k \mid k \text{ is even}\}$$
$$B = \{1^k \mid k \text{ is odd}\}$$

Then $AB$ is the language of strings that consist of an even number of `0`'s followed by an odd number of `1`'s:

$$AB = \{0^i 1^j \mid i \text{ is even and } j \text{ is odd}\}.$$

Here's another example that will demonstrate the usefulness of both the union and concatenation operations. Let $N$ be the language of numbers defined in Exercise 2.2.4: strings that consist of an optional minus sign followed by at least one digit, or an optional minus sign followed by any number of digits, a decimal point and at least one digit. Let $D^*$ denote the language of strings that consist of any number of digits. Let $D^+$ denote the language of strings that consist of at least one digit. Then the language $N$ can be defined as follows:

$$N = \{\varepsilon, -\}D^+ \cup \{\varepsilon, -\}D^*\{.\}D^+.$$

In other words, a language that took 33 words to describe can now defined with a mathematical expression that's about half a line long. In addition, the mathematical expression helps us visualize what the strings of the language look like.

The obvious question now is whether the class of regular languages is closed

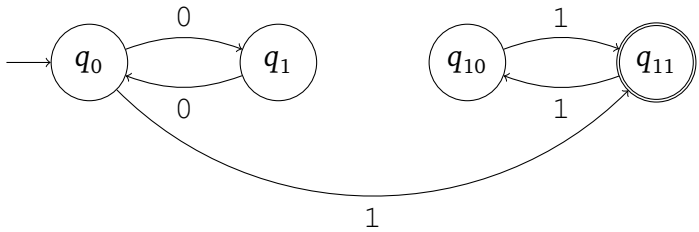



Figure 2.24: A DFA for the language *AB*

under concatenation? And whether we can prove this closure property constructively.

Closure under concatenation is trickier to prove than closure under complementation and closure under union. In many cases, it is easy to design a DFA for the concatenation of two particular languages. For example, Figure 2.24 shows a DFA for the language *AB* mentioned above. (As usual, missing transitions go to a garbage state.) This DFA is essentially a DFA for *A*, on the left, connected to a DFA for *B*, on the right. As soon as the DFA on the left sees a `1` while in its accepting state, the computation switches over to the DFA on the right.

The above example suggests an idea for showing that the concatenation of two regular languages is always regular. Suppose that $M_1$ and $M_2$ are DFA's for *A* and *B*. We want a DFA for *AB* to accept a string in *A* followed by a string in *B*. That is, a string that goes from the start state of $M_1$ to an accepting state of $M_1$, followed by a string that goes from the start state of $M_2$ to an accepting state of $M_2$. So what about we add to each accepting state of $M_1$ all the transitions that come out of the starting state of $M_2$? This is illustrated by Figure 2.25. (The new transitions are shown in a dashed pattern.) This will cause the accepting states of $M_1$ to essentially act as if they were the start state of $M_2$.

But there is a problem with this idea: the accepting states of $M_1$ may now

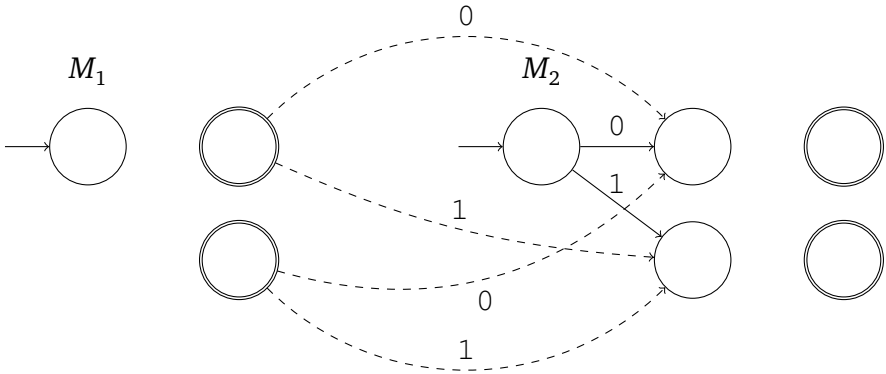



Figure 2.25: An idea for showing that the class of regular languages is closed under concatenation

have multiple transitions labeled by the same symbol. In other words, when the DFA reaches an accepting state of $M_1$, it may not know whether to continue computing in $M_1$ or whether to switch to $M_2$. And this is the key difficulty in proving closure under concatenation: given a string $w$, how can a DFA determine where to split $w$ into $x$ and $y$ so that $x \in A$ and $y \in B$? In the next chapter, we will develop the tools we need to solve this problem.

## Study Questions

2.5.1. What does it mean for a set to be closed under a certain operation?

2.5.2. What is the concatenation of two languages?

## Exercises

2.5.3. Give DFA's for the complement of each of the languages of Exercise 2.3.6.

2.5.4. Each of the following languages is the union or intersection of two simpler languages. In each case, give DFA's for the two simpler languages and then use the pair construction from the proof of Theorem 2.22 to obtain a DFA for the more complex language. In all cases, the alphabet is $\{0, 1\}$.

a) The language of strings of length at least two that have a $1$ in their second position and also contain at least one $0$.

b) The language of strings that contain at least two $1$'s or at least two $0$'s.

c) The language of strings that contain at least two $1$'s and at most one $0$.

d) The language of strings that contain at least two $1$'s and an even number of $0$'s.

2.5.5. Give DFA's for the following languages. In all cases, the alphabet is $\{0, 1, \#\}$.

a) The language of strings of the form $0^i\#1^j$ where $i$ is even and $j \geq 2$.

b) The language of strings of the form $0^i1^j$ where $i$ is even and $j \geq 2$.

# Chapter 3

# Nondeterministic Finite Automata

So far, all our finite automata have been deterministic. This means that each one of their moves is completely determined by the current state and the current input symbol. In this chapter, we learn that finite automata can be nondeterministic. Nondeterministic finite automata are a useful tool for showing that languages are regular. They are also a good introduction to the important concept of nondeterminism.

## 3.1 Introduction

In this section, we introduce the concept of a nondeterministic finite automaton (NFA) through some examples. In the next section, we will formally define what we mean.

Consider again the language of strings that contain the substring `001`. Figure 3.1 shows the DFA we designed in the previous chapter for this language.

Now consider the finite automaton of Figure 3.2. This automaton is not a DFA for two reasons. First, some transitions are missing. For example, state $q_1$

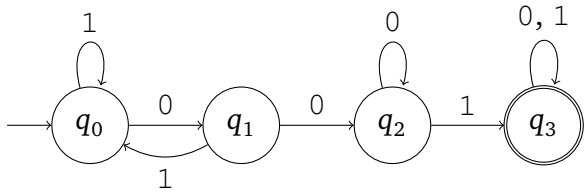



Figure 3.1: A DFA for the language of strings that contain the substring `001`

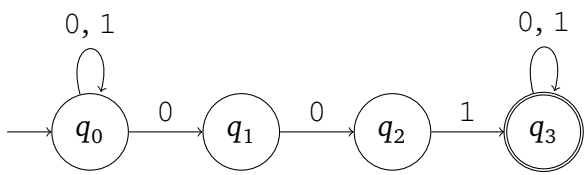



Figure 3.2: An NFA for the language of strings that contain the substring `001`

has no transition labeled 1. If the NFA is in that state and the next input symbol is a 1, we consider that the NFA is stuck. It can't finish reading the input string and is unable to accept it.

Second, some states have multiple transitions for the same input symbol. In the case of this NFA, there are two transitions labeled 0 coming out of state $q_0$. What this means is that when in that state and reading a 0, the NFA has a choice: it can stay in state $q_0$ or move to state $q_1$.

How does the NFA make that choice? We simply consider that if there is an option that eventually leads the NFA to accept the input string, the NFA will choose that option. In this example, if the input string contains the substring 001, the NFA will wait in state $q_0$ until it reaches an occurrence of the substring 001 (there may be more than one), move to the accepting state as it reads the substring 001, and then finish reading the input string while looping in the accepting state.

On the other hand, if the input string does not contain the substring 001, then the NFA will do something, but whatever it does will not allow it to reach the accepting state. That's because to move from the start state to the accepting state requires that the symbols 0, 0 and 1 occur consecutively in the input string.

Here's another example. Consider the language of strings that contain either a number of 0's that's even or a number of 1's that's even. In the previous chapter, we observed that this language is the union of two simpler languages and then used the pair construction to obtain the DFA of Figure 2.23.

An NFA for this language is shown in Figure 3.3. It combines the DFA's of the two simpler languages in a much simpler way than the pair construction.

This NFA illustrates another feature that distinguishes NFA's from DFA's: transitions that are labeled $\varepsilon$. The NFA can use these *$\varepsilon$ transitions* without reading any input symbols. This particular NFA has two $\varepsilon$ transitions, both coming out of the start state. This means that in its start state, the NFA has two options: it

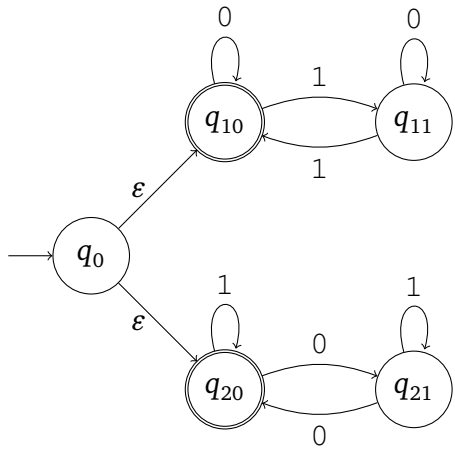



Figure 3.3: An NFA for the language of strings that contain either a number of `0`'s that's even or a number of `1`'s that's even.

can move to either state $q_{10}$ or state $q_{20}$. And it makes that choice before reading the first symbol of the input string.

This NFA operates as described earlier: given multiple options, if there is one that eventually leads the NFA to accept the input string, the NFA will choose that option. In this example, if the number of `1`'s in the input string is even but the number of `0`'s is not, the NFA will choose to move to state $q_{10}$. If instead the number of `0`'s is even but the number of `1`'s is not, the NFA will choose to move to state $q_{20}$. If both the number of `0`'s and the number of `1`'s are even, the NFA will choose to move to either state $q_{10}$ or $q_{20}$, and eventually accept either way. If neither the number of `0`'s nor the number of `1`'s is even, then the NFA will not be able to accept the input string.

Here's one more example. Consider the language of strings of length at least 3 that have a `1` in position 3 from the end. We can design a DFA for this language

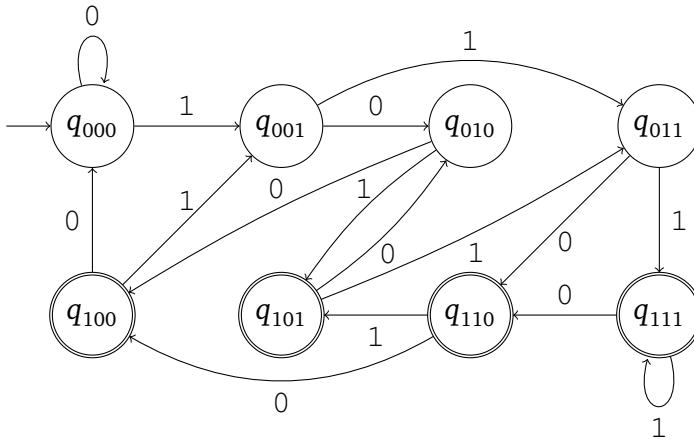



Figure 3.4: A DFA for the language of strings of length at least 3 that have a `1` in position 3 from the end

by having the DFA remember the last three symbols it has seen, as shown in Figure 3.4. The start state is $q_{000}$, which corresponds to assuming that the input string is preceded by `000`. This eliminates special cases while reading the first two symbols of the input string.

Figure 3.5 shows an NFA for the same language. This NFA is surprisingly simpler than the DFA. If the input string does contain a `1` in position 3 from the end, the NFA will wait in the start state until it reaches that `1` and then move to the accepting state as it reads the last three symbols of the input string.

On the other hand, if the input string does not contain a `1` in position 3 from the end, then the NFA will either fall short of reaching the accepting state or it will reach it before having read the last symbol of the input string. In that case, it will be stuck, unable to finish reading the input string. Either way, the input string will not be accepted.

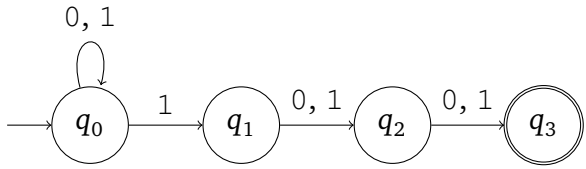



Figure 3.5: An NFA for the language of strings of length at least 3 that have a 1 in position 3 from the end

This example makes two important points. One is that NFA's can be much simpler than DFA's that recognize the same language and, consequently, it can be much easier to design an NFA than a DFA. Second, we consider that an NFA accepts its input string only if it is able to read the entire input string.

To summarize, an NFA is a finite automaton that may have missing transitions, multiple transitions coming out of a state for the same input symbol, and transitions labeled $\varepsilon$ that can be used without reading any input symbol. An NFA accepts a string if there is a way for the NFA to read the entire string and end up in an accepting state.

Earlier, we said that when confronted with multiple options, we consider that the NFA makes the right choice: if there is an option that eventually leads to acceptance, the NFA will choose it. Now, saying that the NFA makes the right choice, if there is one, does not explain *how* the NFA makes that choice. One way to look at this is to simply pretend that the NFA has the magical ability to make the right choice...

Of course, real computers aren't magical. In fact, they're deterministic. So a more realistic way of looking at the computation of an NFA is to imagine that the NFA explores all the possible options, looking for one that will allow it to accept the input string. So it's not that NFA's are magical, it's that they are

somewhat misleading: when we compare the NFA of Figure 3.5 with the DFA of Figure 3.4, the NFA looks much simpler but, in reality, it hides a large amount of computation.

So why are we interested in NFA's if they aren't a realistic model of computation? One answer is that they are useful. We will soon learn that there is a simple algorithm that can convert any NFA into an *equivalent* DFA, that is, one that recognizes the same language. And for some languages, it can be much easier to design an NFA than a DFA. This means that for those languages, it is much easier to design an NFA and then convert it to a DFA than to design the DFA directly.

Another reason for studying NFA's is that they are a good introduction to the concept of nondeterminism. In the context of finite automata, NFA's are no more powerful than DFA's: they do not recognize languages that can't already be recognized by DFA's. But in other contexts, nondeterminism seems to result in additional computational power.

Without going into the details, here's the most famous example. Algorithms are generally considered to be efficient if they run in polynomial time. There is a wide variety of languages that can be recognized by polynomial-time algorithms. But there are also many others that can be recognized by *nondeterministic* polynomial-time algorithms but for which no deterministic polynomial-time algorithm is known. It is widely believed that most of these languages cannot be recognized by deterministic polynomial-time algorithms. After decades of effort, researchers are still unable to prove this but their investigations have led to deep insights into the complexity of computational problems.[1]

[1]What we are referring to here is the famous P vs NP problem and the theory of NP-completeness. Let P be the class of languages that can be recognized by deterministic algorithms that run in polynomial time. Let NP be the class of languages that can be recognized by *nondeterministic* algorithms that run in polynomial time. Then proving that there are languages that can be recognized by nondeterministic polynomial-time algorithms but not by deterministic

## Study Questions

3.1.1. What is an NFA?

3.1.2. What does it mean for an NFA to accept a string?

## Exercises

3.1.3. Give NFA's for the following languages. Each NFA should respect the specified limits on the number of states and transitions. (Transitions labeled with two symbols count as two transitions.) In all cases, the alphabet is $\{0, 1\}$.

a) The language of strings of length at least two that begin with `0` and end in `1`. No more than three states and four transitions.

b) The language of strings of length at least two whose last two symbols are the same. No more than four states and six transitions.

c) The language of strings of length at least two that have a `1` in the second-to-last position. No more than three states and five transitions.

d) The language of strings of length at least $k$ that have a `1` in position $k$ from the end. Do this in general, for every $k \geq 1$. No more than

polynomial-time algorithms is equivalent to showing that P is a strict subset of NP. The consensus among experts is that this is true but no proof has yet been discovered. However, researchers have discovered an amazing connection between a wide variety of languages that belong to NP but are not known to belong to P: if a single one of these languages was shown to belong to P, then every language in NP, all of them, would belong to P. This property is called NP-*completeness*. The fact that a language is NP-complete is considered strong evidence that there are no efficient algorithms that recognize it. The P vs NP problem is considered one of the most important open problems in all of mathematics, as evidenced by the fact that the Clay Mathematics Institute has offered a million dollar prize to the first person who solves it.

$k+1$ states and $2k+1$ transitions. (You did the case $k=2$ in the previous part.)

e) The language of strings that contain exactly one `1`. No more than two states and three transitions.

## 3.2 Formal Definition

In this section, we formally define what an NFA is. As was the case with DFA's, a formal definition is necessary for proving mathematical statements about NFA's and for writing programs that manipulate NFA's.

As explained in the previous section, an NFA is a DFA that can have missing transitions, multiple transitions coming out of a state for the same input symbol, and transitions labeled $\varepsilon$. The first two features can be captured by having the transition function return a set of states: $\delta(q,a)$ will be the set of options available to the NFA from state $q$ when reading symbol $a$. Note that this set can be empty. Transitions labeled $\varepsilon$ can be described by extending the transition function: $\delta(q,\varepsilon)$ will be the set of states that can be reached from state $q$ by $\varepsilon$ transitions.

**Definition 3.1** *A* nondeterministic finite automaton (NFA) *is a* 5*-tuple* $(Q,\Sigma,\delta,q_0,F)$ *where*

1. $Q$ *is a finite set of* states.
2. $\Sigma$ *is an alphabet called the* input alphabet.
3. $\delta : Q \times (\Sigma \cup \varepsilon) \to \mathscr{P}(Q)$ *is the* transition function.
4. $q_0 \in Q$ *is the* starting state.

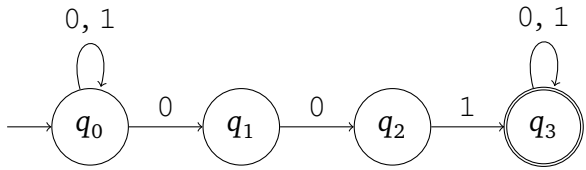



Figure 3.6: An NFA for the language of strings that contain the substring 001

| $\delta$ | 0 | 1 | $\varepsilon$ |
|---|---|---|---|
| $q_0$ | $\{q_0, q_1\}$ | $\{q_0\}$ | $\emptyset$ |
| $q_1$ | $\{q_2\}$ | $\emptyset$ | $\emptyset$ |
| $q_2$ | $\emptyset$ | $\{q_3\}$ | $\emptyset$ |
| $q_3$ | $\{q_3\}$ | $\{q_3\}$ | $\emptyset$ |

Figure 3.7: The transition function of the NFA of Figure 3.6.

5. $F \subseteq Q$ *is the set of* accepting states.

**Example 3.2** Consider the NFA shown in Figure 3.6. Here is the formal description of this NFA: the NFA is $(Q, \{0, 1\}, \delta, q_0, F)$ where

$$Q = \{q_0, q_1, q_2, q_3\}$$
$$F = \{q_3\}$$

and $\delta$ is defined by the table shown in Figure 3.7. □

**Example 3.3** Consider the NFA shown in Figure 3.8. This NFA is

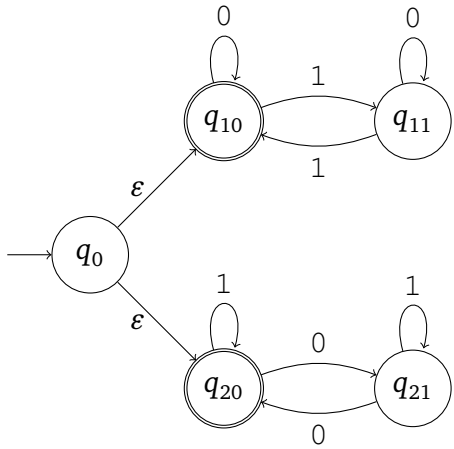



Figure 3.8: An NFA for the language of strings that contain either a number of `0`'s that's even or a number of `1`'s that's even.

$(Q, \{0, 1\}, \delta, q_0, F)$ where

$$Q = \{q_0, q_{10}, q_{11}, q_{20}, q_{21}\}$$
$$F = \{q_{10}, q_{20}\}$$

and $\delta$ is defined by the table shown in Figure 3.9. Note that in this table, we omitted the braces and used a dash (—) instead of the empty set symbol ($\emptyset$). We will often do that to avoid cluttering the tables. □

We now define what it means for an NFA to accept an input string. We first do this for NFA's that don't contain any $\varepsilon$ transitions.

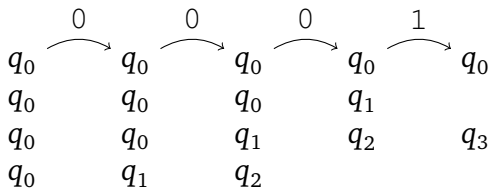
In the case of DFA's (see Definition 2.15), we were able to talk about the sequence of states that the DFA goes through while reading the input string. That was because that sequence was unique. But in the case of NFA's, there may

| $\delta$ | 0 | 1 | $\varepsilon$ |
|---|---|---|---|
| $q_0$ | — | — | $q_{10}, q_{20}$ |
| $q_{10}$ | $q_{10}$ | $q_{11}$ | — |
| $q_{11}$ | $q_{11}$ | $q_{10}$ | — |
| $q_{20}$ | $q_{21}$ | $q_{20}$ | — |
| $q_{21}$ | $q_{20}$ | $q_{21}$ | — |

Figure 3.9: The transition function of the NFA of Figure 3.8.

be multiple sequences of states for each input string, depending on how many options the NFA has.

For example, consider the NFA of Figure 3.6. While reading the string $w =$ 0001, the NFA could go through any of the following four sequences of states:




Two of those sequences end prematurely because the NFA is stuck, unable to read the next input symbol. The first sequence allows the NFA to read the entire input string but it ends in a non-accepting state. We consider that the NFA accepts because one of these four sequences, the third one, allows the NFA to read the entire input string and end in an accepting state.

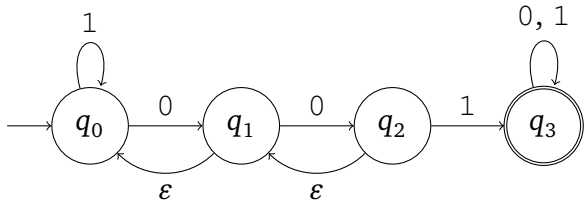



Figure 3.10: Another NFA for the language of strings that contain the substring 001

**Definition 3.4** *Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA without $\varepsilon$ transitions and let $w = w_1 \cdots w_n$ be a string of length $n$ over $\Sigma$. Then $N$* accepts *$w$ if and only if there is a sequence of states $r_0, r_1, \ldots, r_n$ such that*

$$r_0 = q_0 \tag{3.1}$$

$$r_i \in \delta(r_{i-1}, w_i), \quad \text{for } i = 1, \ldots, n \tag{3.2}$$

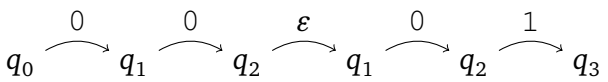
$$r_n \in F. \tag{3.3}$$

Equations 3.1 and 3.2 assert that the sequence of states $r_0, r_1, \ldots, r_n$ is one of the possible sequences of states that the NFA may go through while reading (all of) $w$. Equation 3.3 says that this sequence of states leads to an accepting state.

We now define acceptance for NFA's that may contain $\varepsilon$ transitions. This is a little trickier. Figure 3.10 shows another NFA for the language of strings that contain the substring 001. Consider again the string $w = 0001$. The NFA accepts this string because it could go through the following sequence of states as it reads $w$:


For the NFA to go through this sequence of states, the NFA needs to essentially "insert" an $\varepsilon$ between the first and second symbols of $w$. That is, it needs to view $w$ as $0\varepsilon001$.

Another possible sequence of states is




In this case, the NFA inserts an $\varepsilon$ between the second and third symbols of $w$. That is, it views $w$ as $00\varepsilon01$.

A definition of acceptance for NFA's with $\varepsilon$ transitions can be based on this idea of inserting $\varepsilon$ into the input string. Note that inserting $\varepsilon$ into a string does not change its "value"; that is, as strings, we have that $0001 = 0\varepsilon001 = 00\varepsilon01$. (In fact, for every string $x$, $x\varepsilon = \varepsilon x = x$. We say that $\varepsilon$ is an neutral element with respect to concatenation of strings.)

**Definition 3.5** *Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and let $w = w_1 \cdots w_n$ be a string of length $n$ over $\Sigma$. Then $N$* accepts *$w$ if and only if $w$ can be written as $y_1 \cdots y_m$, with $y_i \in \Sigma \cup \{\varepsilon\}$, for every $i = 1, \ldots, m$, and there is a sequence of states $r_0, r_1, \ldots, r_m$ such that*

$$r_0 = q_0 \tag{3.4}$$

$$r_i \in \delta(r_{i-1}, y_i), \quad \text{for } i = 1, \ldots, m \tag{3.5}$$

$$r_m \in F. \tag{3.6}$$

Note that in this definition, $m$ could be greater than $n$. In fact, $m - n$ equals the number of $\varepsilon$ transitions that the NFA needs to use to go through the sequence

of states $r_0, \ldots, r_m$ as it reads $w$. Once again, Equations 3.4 and 3.5 assert that this sequence of states is one of the possible sequences of states the NFA may go through while reading $w$, and Equation 3.6 says that this sequence of states leads to an accepting state.

### Exercises

3.2.1. Give formal descriptions of the following NFA's. In each case, describe the transition function by using a transition table.

a) The NFA of Figure 3.5.

b) The NFA of Figure 3.10.

3.2.2. Consider the NFA of Figure 3.6. There are three possible sequences of states that this NFA could go through while reading all of the string `001001`. What are they? Does the NFA accept this string?

3.2.3. Consider the NFA of Figure 3.10. There are seven possible sequences of states that this NFA could go through while reading all of the string `001001`. What are they? Indicate where $\varepsilon$'s need to be inserted into the string. Does the NFA accept this string?

## 3.3 Equivalence with DFA's

Earlier in this chapter, we mentioned that there is a simple algorithm that can convert any NFA into an *equivalent* DFA, that is, one that recognizes the same language. In this section, we are going to learn what this algorithm is. Since DFA's are just a special case of NFA's, this will show that DFA's and NFA's recognize the same class of languages.

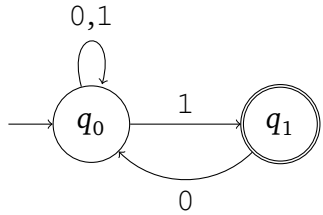



Figure 3.11: An NFA for the language of strings that end in 1

Consider the NFA of Figure 3.11. This NFA recognizes the language of strings that end in 1. It's not the simplest possible NFA for this language, but if an algorithm is going to be able to convert NFA's into DFA's, it has to be able to handle any NFA.

Now consider the input string $w = 1101101$. The diagram of Figure 3.12 shows all the possible ways in which the NFA could process this string. This diagram is called the *computation tree* of the NFA on string $w$. The nodes in this tree are labeled by states. The root of the tree (shown at the top) represents the beginning of the computation. Edges show possible moves that the NFA can make. For example, from the start state, the NFA has two options when reading symbol 1: stay in state $q_0$ or go to state $q_1$.

Some nodes in the computation tree are dead ends. For example, the node labeled $q_1$ at level 1 is a dead end because there is no 1 transition coming out of state $q_1$. (Note that the top level of a tree is considered to be level 0.)

The nodes at each level of the computation tree show all the possible states that the NFA could be in after reading a certain portion of the input string. For example, level 4 has nodes labeled $q_0, q_1, q_0, q_1$. These are the states that the NFA could be in after reading 1101.

The nodes at the bottom level show the states that the NFA could be in after

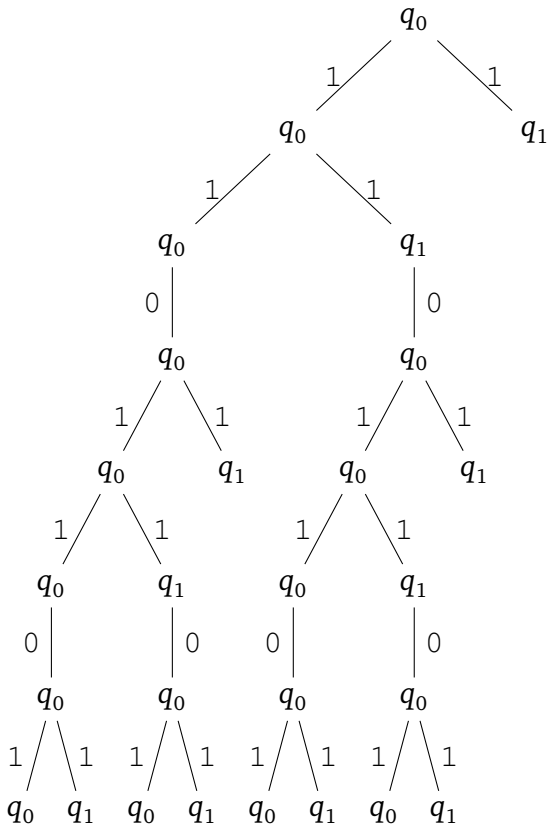



Figure 3.12: The computation tree of the NFA of Figure 3.11 on input string 1101101

reading the entire string. We can see that this NFA accepts this input string because the bottom level contains the accepting state $q_1$.

Computation trees help us visualize all the possible ways in which an NFA can process a particular input string. But they also suggest a way of simulating NFA's: as we read the input string, we can move down the computation tree, figuring out what nodes occur at each of its levels. And we don't need to remember the entire tree, only the nodes at the current level, the one that corresponds to the last input symbol that was read.[2] For example, in the computation tree of Figure 3.12, after reading the string `1101`, we would have figured out that the current level of the computation tree consists of states $q_0, q_1, q_0, q_1$.

How would a DFA carry out this simulation? In the only other simulation we have seen so far, the pair construction of Theorem 2.22, the DFA $M$ simulates the DFA's $M_1$ and $M_2$ by keeping track of the state in which each of these DFA's would be. Each of the states of $M$ is a pair that combined a state of $M_1$ with a state of $M_2$.

In our case, we would have each state of the DFA that simulates an NFA be the sequence of states that appear at the current level of the computation tree of the NFA on the input string. However, as the computation tree of Figure 3.12 clearly indicates, the number of nodes at each level can grow. For example, if an NFA has two options at each move, which is certainly possible, then the bottom level of the tree would contain $2^n$ nodes, where $n$ is the length of the input string. This implies that there may be an infinite number of possible sequences of states that can appear at any level of the computation tree. Since the DFA has a finite number of states, it can't have a state for each possible sequence.

A solution to this problem comes from noticing that most levels of the tree

[2] Some of you may recognize that this is essentially a breadth-first search. At Clarkson, this algorithm and other important graph algorithms are normally covered in the courses CS344 *Algorithms and Data Structures* and CS447 *Computer Algorithms*.

of Figure 3.12 contain a lot of repetition. And we only want to determine if an accepting state occurs at the bottom level of the tree, not how many times it occurs, or how many different ways it can be reached from the start state. Therefore, as it reads the input string, the DFA only needs to remember the *set* of states that occur at the current level of the tree (without repetition). This corresponds to pruning the computation tree by eliminating duplicate subtrees, as shown in Figure 3.13. The pruning locations are indicated by dots ($\cdots$).

Another way of looking at this is to say that as it reads the input string, the DFA simulates the NFA by keeping track of the set of states that the NFA could currently be in. If the NFA has $k$ states, then there are $2^k$ different sets of states. Since $k$ is a constant, $2^k$ is also a constant. This implies that the DFA can have a state for each set of states of the NFA.

Here are the details of the simulation, first for the case of NFA's without $\varepsilon$ transitions.

**Theorem 3.6** *Every NFA without $\varepsilon$ transitions has an equivalent DFA.*

**Proof** Suppose that $N = (Q, \Sigma, \delta, q_0, F)$ is an NFA without $\varepsilon$ transitions. As explained above, we construct a DFA $M$ that simulates $N$ by keeping track of the set of states that $N$ could currently be in. More precisely, each state of $M$ will be a set of states of $N$. If after reading a string $w$, $R$ is the set of states that $N$ could be in, then $M$ will be in state $R$.

We need to specify the start state, the accepting states and the transition function of $M$. Initially, $N$ can only be in its start state so the start state of $M$ will be $\{q_0\}$.

NFA $N$ accepts a string $w$ if and only if the set of states it could be in after reading $w$ includes at least one accepting state. Therefore, the accepting states of $M$ will be those sets $R$ that contain at least one state from $F$.

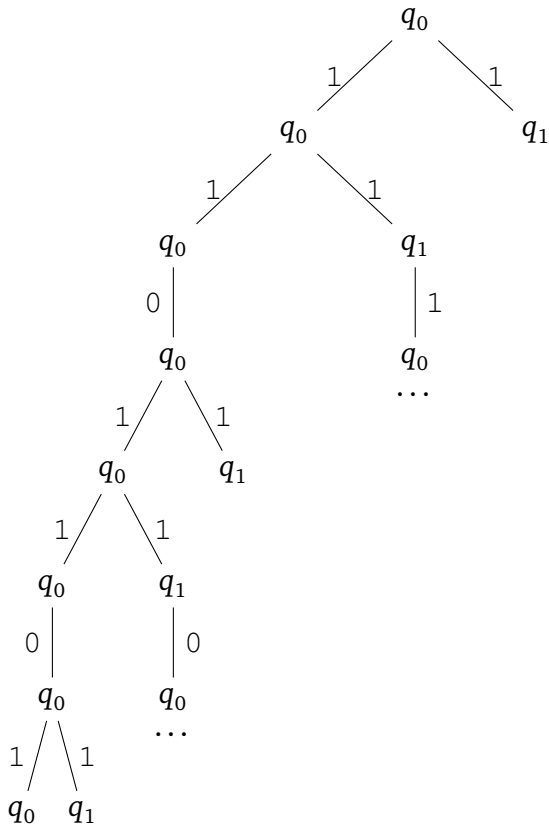



Figure 3.13: The computation tree of Figure 3.12 with duplicate subtrees removed

If $R$ is the set of states that $N$ can currently be in, then after reading one more symbol $a$, $N$ could be in any state that can be reached from a state in $R$ by a transition labeled $a$. That is, $N$ could be in any state in the following set:

$$\{q \in Q \mid q \in \delta(r, a), \text{ for some } r \in R\}.$$

Therefore, from state $R$, $M$ will have a transition labeled $a$ going to the state that corresponds to that set.

To summarize, the DFA $M$ is $(Q', \Sigma, \delta', q_0', F')$ where

$$\begin{aligned} Q' &= \mathscr{P}(Q) \\ q_0' &= \{q_0\} \\ F' &= \{R \subseteq Q \mid R \cap F \neq \emptyset\} \end{aligned}$$

and $\delta'$ is defined as follows: for every $R \subseteq Q$ and $a \in \Sigma$,

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

From the above description of $M$, it should be clear that $L(M) = L(N)$. □

**Example 3.7** Let's construct a DFA that simulates the NFA of Figure 3.11. We get $M = (Q', \{0, 1\}, \delta', q_0', F')$ where

$$\begin{aligned} Q' &= \{\emptyset, \{q_0\}, \{q_1\}, \{q_0, q_1\}\} \\ q_0' &= \{q_0\} \\ F' &= \{\{q_1\}, \{q_0, q_1\}\} \end{aligned}$$

and $\delta'$ is defined by the table shown in Figure 3.14.

The entries in this table were computed as follows. First, $\delta'(\emptyset, a) = \emptyset$ because

| $\delta'$ | 0 | 1 |
|---|---|---|
| $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $\{q_0\}$ | $\{q_0\}$ | $\{q_0, q_1\}$ |
| $\{q_1\}$ | $\{q_0\}$ | $\emptyset$ |
| $\{q_0, q_1\}$ | $\{q_0\}$ | $\{q_0, q_1\}$ |

Figure 3.14: The transition function of the DFA that simulates the NFA of Figure 3.11

an empty union gives an empty set. (This is consistent with the fact that no states can be reached from any of the states in an empty set.) Then $\delta'(\{q_0\}, a) = \delta(q_0, a)$ and similarly for $\{q_1\}$. Finally,

$$\delta'(\{q_0, q_1\}, a) = \delta(q_0, a) \cup \delta(q_1, a) = \delta(\{q_0\}, a) \cup \delta(\{q_1\}, a)$$

so that the $\delta'(\{q_0, q_1\}, a)$ values can be computed from the others.

Figure 3.15 shows the transition diagram of the DFA. Note that state $\{q_1\}$ cannot be reached from the start state. This means that it can be removed from the DFA. And if state $\{q_1\}$ is removed, then state $\emptyset$ also becomes unreachable from the start state. So it can be removed too. This leaves us with the DFA of Figure 3.16. Note that apart from the names of the states, this DFA is identical to the DFA that we directly designed in the previous chapter (see Figure 2.12). □

We now tackle the case of NFA's with $\varepsilon$ transitions. The DFA now needs to account for the fact that the NFA may use any number of $\varepsilon$ transitions between any two input symbols, as well as before the first symbol and after the last symbol.

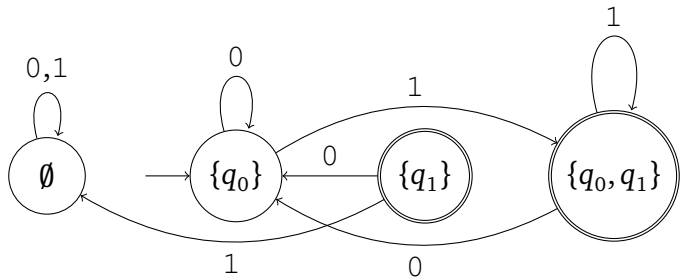



Figure 3.15: The DFA that simulates the NFA of Figure 3.11

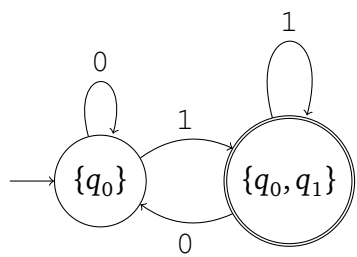



Figure 3.16: A simplified DFA that simulates the NFA of Figure 3.11

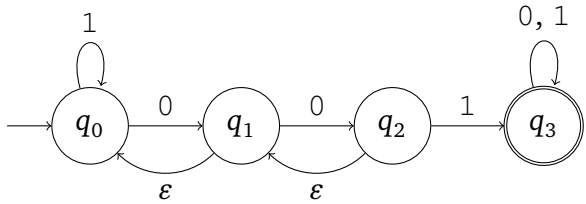



Figure 3.17: An NFA for the language of strings that contain the substring `001`

The following concept will be useful.

**Definition 3.8** *The* extension *of a set of states R, denoted $E(R)$, is the set of states that can be reached from any state in R by using any number of $\varepsilon$ transitions (none included).*[3]

**Example 3.9** Figure 3.17 shows an NFA for the language of strings that contain the substring `001`. In this NFA, we have the following:

$$E(\{q_0\}) = \{q_0\}$$
$$E(\{q_1\}) = \{q_0, q_1\}$$
$$E(\{q_2\}) = \{q_0, q_1, q_2\}$$

[3]This can be defined more formally as follows: $E(R)$ is the set of states $q$ for which there is a sequence of states $r_0, \ldots, r_k$, for some $k \geq 0$, such that

$$r_0 \in R$$
$$r_i \in \delta(r_{i-1}, \varepsilon), \quad \text{for } i = 1, \ldots, k$$
$$r_k = q.$$

In addition,

$$E(\{q_0, q_1\}) = E(\{q_0\}) \cup E(\{q_1\}) = \{q_0\} \cup \{q_0, q_1\} = \{q_0, q_1\}.$$

This illustrates how the extension of a set that contains more than one state can be computed as the union of the extensions of singletons (sets that contain exactly one state). □

**Theorem 3.10** *Every NFA has an equivalent DFA.*

**Proof** Suppose that $N = (Q, \Sigma, \delta, q_0, F)$ is an NFA. We construct a DFA $M$ that simulates $N$ pretty much as in the proof of the previous theorem. In particular, if $R$ is the set of states that $N$ could be in after reading a string $w$ and using any number of $\varepsilon$ transitions, then $M$ will be in state $R$.

One difference from the previous theorem is that $M$ will not start in state $\{q_0\}$ but in state $E(\{q_0\})$. This will account for the fact that $N$ can use any number of $\varepsilon$ transitions even before reading the first input symbol.

The other difference is that from state $R$, $M$ will have a transition labeled $a$ going to state

$$\{q \in Q \mid q \in E(\delta(r, a)), \text{ for some } r \in R\}.$$

This will account for the fact that $N$ can use any number of $\varepsilon$ transitions after reading each input symbol (including the last one).

This is what we get: $M = (Q', \Sigma, \delta', q_0', F')$ where

$$\begin{aligned} Q' &= \mathscr{P}(Q) \\ q_0' &= E(\{q_0\}) \\ F' &= \{R \subseteq Q \mid R \cap F \neq \emptyset\} \end{aligned}$$

and $\delta'$ is defined as follows:

$$\delta'(R,a) = \bigcup_{r \in R} E(\delta(r,a)), \quad \text{for } R \subseteq Q \text{ and } a \in \Sigma.$$

It should be clear that $L(M) = L(N)$. □

**Example 3.11** Let's construct a DFA that simulates the NFA of Figure 3.17. The easiest way to do this by hand is usually to start with the transition table and proceed as follows:

1. Compute $\delta'(\{r\},a) = E(\delta(r,a))$ for the individual states $r$ of the NFA. Those values are shown in the top half of Figure 3.18.

2. Identify the start state. In this case it's $E(\{q_0\}) = \{q_0\}$.

3. Add states as needed (to the bottom half of the table) until no new states are introduced. The value of the transition function for these states is computed from the values in the top half of the table by taking advantage of the following fact:

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

Note that in this table, we omitted the braces and used a dash (—) instead of the empty set symbol (∅).

All that we have left to do now is to identify the accepting states of the DFA. In this case, they're the sets of states that contain $q_3$, the accepting state of the NFA.

Figure 3.19 shows the transition diagram of the DFA. Note that the three accepting states can be merged since all the transitions coming out of these states

| $\delta'$ | 0 | 1 |
|---|---|---|
| $q_0$ | $q_0, q_1$ | $q_0$ |
| $q_1$ | $q_0, q_1, q_2$ | $-$ |
| $q_2$ | $-$ | $q_3$ |
| $q_3$ | $q_3$ | $q_3$ |
| $q_0, q_1$ | $q_0, q_1, q_2$ | $q_0$ |
| $q_0, q_1, q_2$ | $q_0, q_1, q_2$ | $q_0, q_3$ |
| $q_0, q_3$ | $q_0, q_1, q_3$ | $q_0, q_3$ |
| $q_0, q_1, q_3$ | $q_0, q_1, q_2, q_3$ | $q_0, q_3$ |
| $q_0, q_1, q_2, q_3$ | $q_0, q_1, q_2, q_3$ | $q_0, q_3$ |

Figure 3.18: The transition function of the DFA that simulates the NFA of Figure 3.17

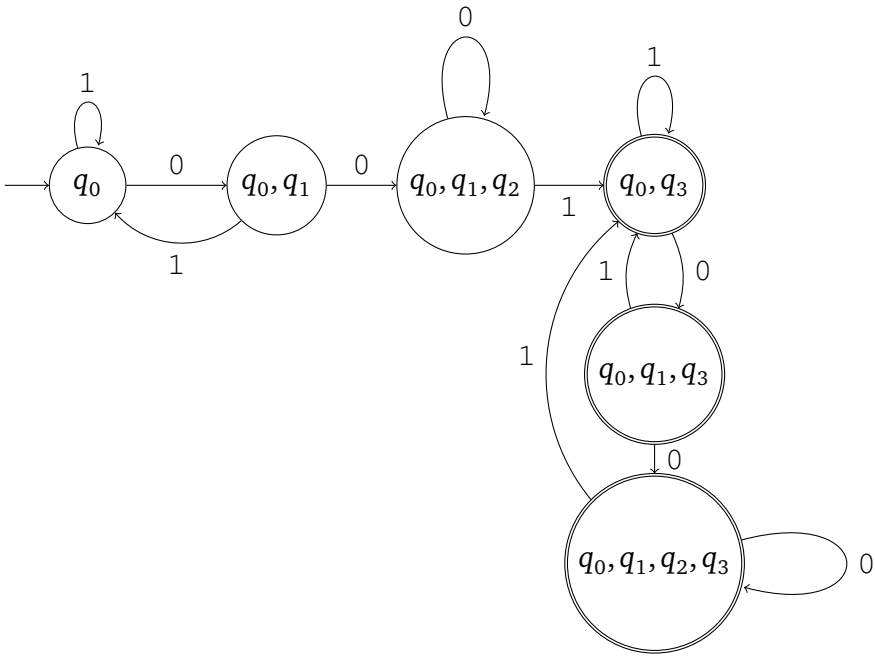



Figure 3.19: A DFA that simulates the NFA of Figure 3.17

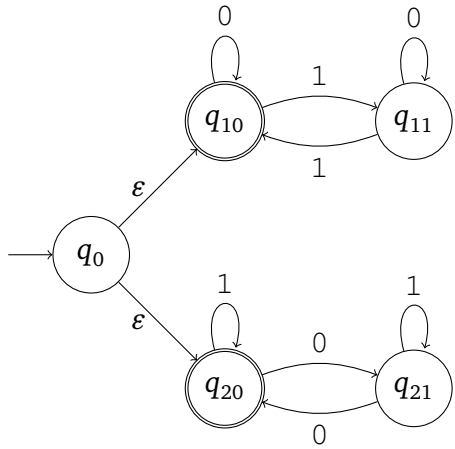



Figure 3.20: An NFA for the language of strings that contain either a number of 0's that's even or a number of 1's that's even.

stay within this group of states. If we merged these states, then this DFA would be identical to the DFA we designed in the previous chapter (see Figure 2.14). □

**Example 3.12** Let's construct a DFA that simulates the NFA of Figure 3.20. Figure 3.21 shows the transition table of the DFA. The start state is $E(\{q_0\}) = \{q_0, q_{10}, q_{20}\}$, which is why this state is the first one that appears in the bottom half of the table. The accepting states are all the states that contain either $q_{10}$ or $q_{20}$, the two accepting states of the NFA.

Note that states $\{q_0, q_{10}, q_{20}\}$ and $\{q_{10}, q_{20}\}$ could be merged since they are both accepting and their transitions lead to exactly the same states. If we did that, then we would find that this DFA is identical to the DFA we designed in the previous chapter by using the pair construction algorithm (see Figure 2.23).

By the way, since the NFA has 5 states, in principle, the DFA has $2^5 = 32$

| $\delta'$ | 0 | 1 |
|---|---|---|
| $q_0$ | — | — |
| $q_{10}$ | $q_{10}$ | $q_{11}$ |
| $q_{11}$ | $q_{11}$ | $q_{10}$ |
| $q_{20}$ | $q_{21}$ | $q_{20}$ |
| $q_{21}$ | $q_{20}$ | $q_{21}$ |
| $q_0, q_{10}, q_{20}$ | $q_{10}, q_{21}$ | $q_{11}, q_{20}$ |
| $q_{10}, q_{21}$ | $q_{10}, q_{20}$ | $q_{11}, q_{21}$ |
| $q_{11}, q_{20}$ | $q_{11}, q_{21}$ | $q_{10}, q_{20}$ |
| $q_{10}, q_{20}$ | $q_{10}, q_{21}$ | $q_{11}, q_{20}$ |
| $q_{11}, q_{21}$ | $q_{11}, q_{20}$ | $q_{10}, q_{21}$ |

Figure 3.21: The transition function of the DFA that simulates the NFA of Figure 3.20

states. But as the transition table indicates, there are only 5 states that are reachable from the start state. □

In this section, we used the technique described in the proofs of Theorems 3.6 and 3.10 to convert three NFA's into equivalent DFA's. Does that mean that these proofs are constructive? Yes but with one caveat: in the proof of Theorem 3.10, we didn't specify how to compute the extension of a set of states. This is essentially what is called a graph reachability problem. We will learn a graph reachability algorithm later in these notes.

Another observation. A language is regular if it is recognized by some DFA. This is the definition. But now we know that every NFA can be simulated by a DFA. We also know that a DFA is a special case of an NFA. Therefore, we get the following alternate characterization of regular languages:

**Corollary 3.13** *A language is regular if and only if it is recognized by some NFA.*

## Study Questions

3.3.1. In an NFA, what is the extension of a state?

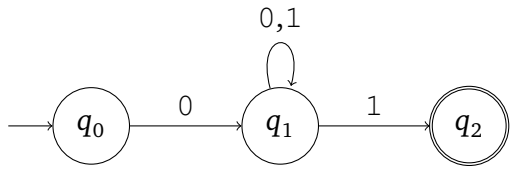
## Exercises

3.3.2. Draw the computation tree of the NFA of Figure 3.6 on the input string `001001`. Prune as needed.

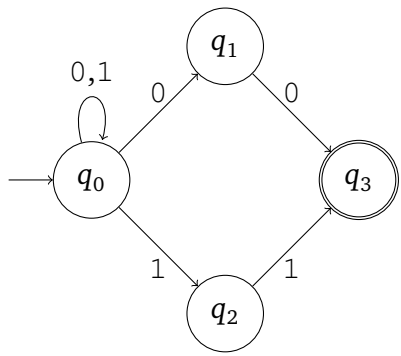
3.3.3. By using the algorithm of this section, convert into DFA's the NFA's of Figures 3.22 and 3.23.

3.3.4. By using the algorithm of this section, convert into DFA's the NFA's of Figure 3.24.

a)




b)




c)

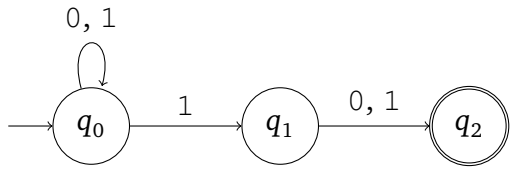



Figure 3.22: NFA's for Exercise 3.3.3 (part 1 of 2)

d)

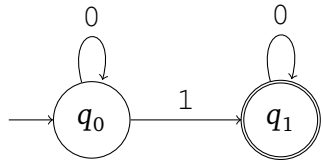
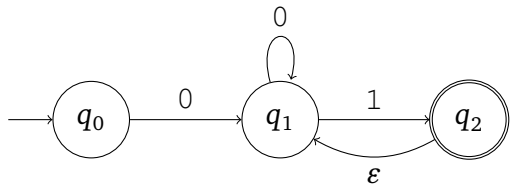



Figure 3.23: NFA's for Exercise 3.3.3 (part 2 of 2)

a)




b)

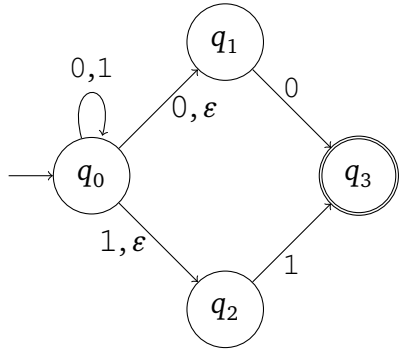



Figure 3.24: NFA's for Exercise 3.3.4

## 3.4 Closure Properties

In Section 2.5, we proved that the class of regular languages is closed under complementation, union and intersection. We also discussed closure under concatenation, realized that this was more difficult to establish, and announced that we would learn the necessary tools in this chapter.

Well, that tool is the NFA. We saw a hint of that in the NFA of Figure 3.3. That NFA recognized the union of two languages and was designed as a combination of DFA's for those two languages. Now that we know that every NFA can be simulated by a DFA, this gives us another way of showing that the class of regular languages is closed under union.

**Theorem 3.14** *The class of regular languages is closed under union.*

**Proof** Suppose that $A_1$ and $A_2$ are regular and let $M_1$ and $M_2$ be DFA's for these languages. We construct an NFA $N$ that recognizes $A_1 \cup A_2$.

The idea is illustrated in Figure 3.25. We add to $M_1$ and $M_2$ a new start state that we connect to the old start states with $\varepsilon$ transitions. This gives the NFA the option of processing the input string by using either $M_1$ or $M_2$. If $w \in A_1 \cup A_2$, then $N$ will choose the appropriate DFA and accept. And if $N$ accepts, it must be that one of the DFA's accepts $w$. Therefore, $L(N) = A_1 \cup A_2$.

The NFA can be described more precisely as follows. Suppose that $M_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$, for $i = 1, 2$. Without loss of generality, assume that $Q_1$ and $Q_2$ are disjoint. (Otherwise, rename the states so the two sets become disjoint.) Let $q_0$ be a state not in $Q_1 \cup Q_2$. Then $N = (Q, \Sigma, \delta, q_0, F)$ where

$$Q = Q_1 \cup Q_2 \cup \{q_0\}$$
$$F = F_1 \cup F_2$$

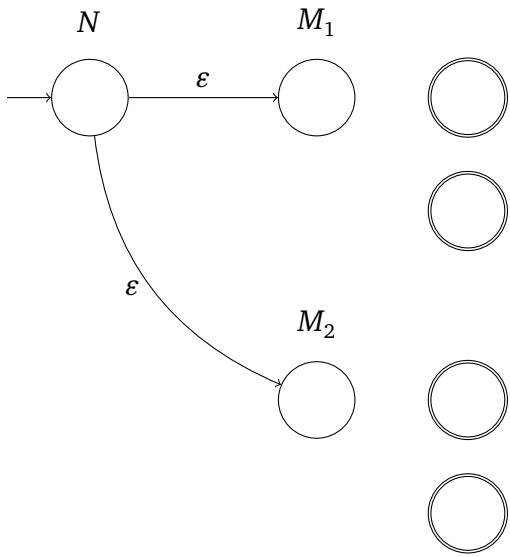



Figure 3.25: An NFA for the union of two regular languages

and $\delta$ is defined as follows:

$$\delta(q,\varepsilon) = \begin{cases} \{q_1, q_2\} & \text{if } q = q_0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\delta(q,a) = \begin{cases} \{\delta_i(q,a)\} & \text{if } q \in Q_i \text{ and } a \in \Sigma \\ \emptyset & \text{if } q = q_0 \text{ and } a \in \Sigma. \end{cases}$$

□

In the previous chapter, we proved closure under union by using the pair construction. And we observed that the construction could be easily modified to prove closure under intersection. We can't do this here: there is no known simple way to adapt the above construction for the case of intersection. But we can still easily prove closure under intersection by using De Morgan's Law.

**Corollary 3.15** *The class of regular languages is closed under intersection.*

**Proof** Suppose that $A_1$ and $A_2$ are two languages. Then

$$A_1 \cap A_2 = \overline{\overline{A_1} \cup \overline{A_2}}.$$

If $A_1$ and $A_2$ are regular, then $\overline{A_1}$ and $\overline{A_2}$ are regular, $\overline{A_1} \cup \overline{A_2}$ is regular, and then so is $\overline{\overline{A_1} \cup \overline{A_2}}$. This implies that $A_1 \cap A_2$ is regular. □

Let's now turn to concatenation. Figure 2.25 illustrates one possible idea: connect DFA's $M_1$ and $M_2$ *in series* by adding transitions so that the accepting states of $M_1$ act as the start state of $M_2$. By using $\varepsilon$ transitions, we can simplify this a bit.

**Theorem 3.16** *The class of regular languages is closed under concatenation.*

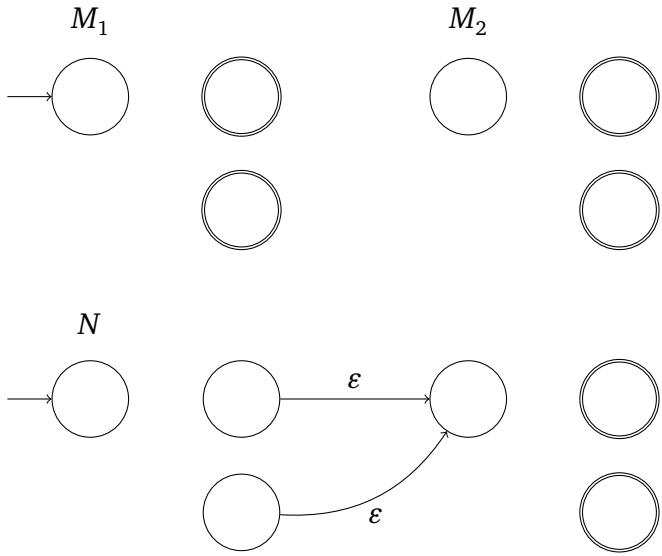



Figure 3.26: An NFA for the concatenation of two regular languages

**Proof** Suppose that $A_1$ and $A_2$ are regular and let $M_1$ and $M_2$ be DFA's for these languages. We construct an NFA $N$ that recognizes $A_1A_2$.

The idea is illustrated in Figure 3.26. We add to the accepting states of $M_1$ $\varepsilon$ transitions to the start state of $M_2$. This gives $N$ the option of "switching" to $M_2$ every time it enters one of the accepting states of $M_1$. We also make the accepting states of $M_1$ non-accepting.

Let's make sure this really works. Suppose that $w \in A_1A_2$. That is, $w = xy$ with $x \in A_1$ and $y \in A_2$. Then after reading $x$, $N$ will be in one of the accepting states of $M_1$. From there, it can use one of the new $\varepsilon$ transitions to move to the start state of $M_2$. The string $y$ will then take $N$ to one of the accepting states of $M_2$, causing $N$ to accept $w$.

Conversely, if $w$ is accepted by $N$, it must be that $N$ uses one of the new $\varepsilon$ transitions. This means that $w = xy$ with $x \in A_1$ and $y \in A_2$. Therefore, $L(N) = A_1A_2$.

The formal description of $N$ is left as an exercise. □

We end this section by considering another operation on languages. If $A$ is a language over $\Sigma$, then the *star* of $A$ is the language

$$A^* = \{x_1 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in A\}.$$

That is, $A^*$ consists of those strings we can form, in all possible ways, by taking any number of strings from $A$ and concatenating them together.[4] Note that $\varepsilon$ is always in the star of a language because, by convention, $x_1 \cdots x_k = \varepsilon$ when $k = 0$.

For example, $\{0\}^*$ is the language of all strings that contain only 0's: $\{\varepsilon, 0, 00, 000, \ldots\}$. And $\{0,1\}^*$ is the language of all strings that can be formed with 0's and 1's: $\{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. Note that this definition of the star operation is consistent with our use of $\Sigma^*$ to denote the set of all strings over the alphabet $\Sigma$.

**Theorem 3.17** *The class of regular languages is closed under the star operation.*

**Proof** Suppose that $A$ is regular and let $M$ be a DFA that recognizes this languages. We construct an NFA $N$ that recognizes $A^*$.

The idea is illustrated in Figure 3.27. From each of the accepting states of $M$, we add $\varepsilon$ transitions that loop back to the start state.

We also need to ensure that $\varepsilon$ is accepted by $N$. One idea is to simply make the start state of $M$ an accepting state. But this doesn't work, as one of the

[4] The language $A^*$ is also sometimes called the *Kleene closure* of $A$.

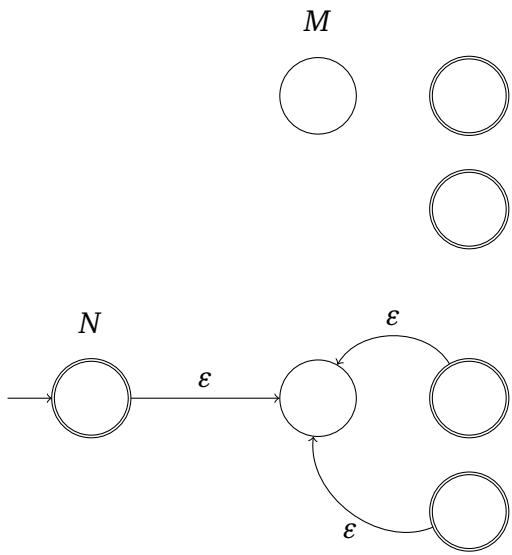



Figure 3.27: An NFA for the star of a regular language

exercises asks you to demonstrate. A better idea is to add a new start state with an $\varepsilon$ transition to the old start state and make the new start state an accepting state.

If $w = x_1 \cdots x_k$ with each $x_i \in A$, then $N$ can accept $w$ by going through $M$ $k$ times, each time reading one $x_i$ and then returning to the start state of $M$ by using one of the new $\varepsilon$ transitions (except after $x_k$).

Conversely, if $w$ is accepted by $N$, then it must be that either $w = \varepsilon$ or that $N$ uses the new $\varepsilon$ "looping back" transitions $k$ times, for some number $k \geq 0$, breaking $w$ up into $x_1 \cdots x_{k+1}$, with each $x_i \in A$. In either case, this implies that $w \in A^*$. Therefore, $L(N) = A^*$.

The formal description of $N$ is left as an exercise. □

## Study Questions

3.4.1. What is the star of a language?

## Exercises

3.4.2. Give a formal description of the NFA of the proof of Theorem 3.16.

3.4.3. Give a formal description of the NFA of the proof of Theorem 3.17.

3.4.4. The proof of Theorem 3.17 mentions the idea illustrated in Figure 3.28. From each of the accepting states of $M$, we add $\varepsilon$ transitions that loop back to the start state. In addition, we make the start state of $M$ accepting to ensure that $\varepsilon$ is accepted.

a) Explain where the proof of the theorem would break down if this idea was used.

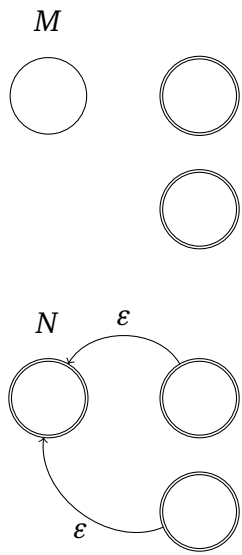



Figure 3.28: A bad idea for an NFA for the star of a regular language

b) Show that this idea cannot work by providing an example of a DFA $M$ for which the NFA $N$ of Figure 3.28 would not recognize $L(M)^*$.

3.4.5. For every language $A$ over alphabet $\Sigma$, let

$$A^+ = \{x_1 \cdots x_k \mid k \geq 1 \text{ and each } x_i \in \Sigma^*\}.$$

Show that the class of regular languages is closed under the "plus" operation.

3.4.6. Show that a language is regular if and only if it can be recognized by some NFA with exactly one accepting state.

3.4.7. If $w = w_1 \cdots w_n$ is a string of length $n$, let $w^{\mathcal{R}}$ denote the reverse of $w$, that is, the string $w_n \cdots w_1$. The reverse $L^{\mathcal{R}}$ of a language $L$ is defined as the language of strings $w^{\mathcal{R}}$ where $w \in L$. Show that the class of regular languages is closed under reversal, that is, show that if $L$ is regular, then $L^{\mathcal{R}}$ is also regular.

# Chapter 4

# Regular Expressions

In the previous two chapters, we studied two types of machines that recognize languages. In this chapter, we approach languages from a different angle: instead of recognizing them, we will describe them. We will learn that regular expressions allow us to describe languages precisely and, often, concisely. We will also learn that regular expressions are powerful enough to describe any regular language, that regular expressions can be easily converted to DFA's, and that it is often easier to write a regular expression than to directly design a DFA or an NFA. This implies that regular expressions are useful not only for describing regular languages but also as a tool for obtaining DFA's for these languages.

## 4.1 Introduction

Many of you are probably already familiar with regular expressions. For example, in the Unix or Linux operating systems, when working at the prompt (on a console or terminal) we can list all the PDF files in the current directory (folder) by using the command `ls *.pdf`. The string `ls` is the name of the command.

(It's short for *list*.) The string `*.pdf` is a regular expression. In Unix regular expressions, the star (`*`) represents any string. So this command is asking for a list of all the files whose name consists of any string followed by the characters `.pdf`.

Another example is `rm project1.*`. This removes all the files associated with Project 1, that is, all the files that have the name `project1` followed by any extension.

Regular expressions are convenient. They allow us to precisely and concisely describe many interesting patterns in strings. But note that a regular expression corresponds to a *set* of strings, those strings that possess the pattern. Therefore, regular expressions describe languages.

In the next section, we will define precisely what we mean by a regular expression. Our regular expressions will be a little different from Unix regular expressions. In the meantime, here are two examples that illustrate both the style of regular expressions we will use as well as the usefulness of regular expressions.

**Example 4.1** Consider the language of valid C++ identifiers. Recall that these are strings that begin with an underscore or a letter followed by any number of underscores, letters and digits. Figure 4.1 shows the DFA we designed earlier for this language. This DFA is not very complicated but a regular expression is even simpler. Let $D$ stand for any digit and $L$ for any letter. Then the valid C++ identifiers can be described as follows:

$$(\_ \cup L)(\_ \cup L \cup D)^*$$

This expression says that an identifier is an underscore or a letter followed by any number of underscores, letters and digits. Note how the star is used not to represent any string but as an *operator* that essentially means "any number of".

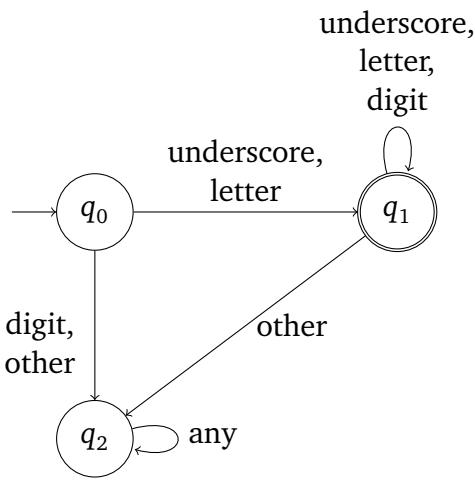



Figure 4.1: A DFA for the set of valid C++ identifiers

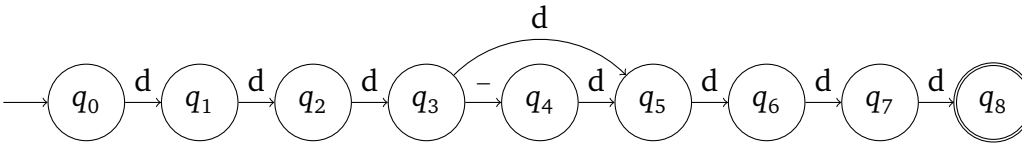



Figure 4.2: A DFA for a language of phone numbers

We can make the regular expression more precise be defining what we mean with the symbols $L$ and $D$. Here too, regular expressions can be used:

$$L = \mathtt{a} \cup \mathtt{b} \cup \mathtt{c} \cup \cdots \cup \mathtt{Z}$$
$$D = \mathtt{0} \cup \mathtt{1} \cup \mathtt{2} \cup \cdots \cup \mathtt{9}$$

□

**Example 4.2** Consider the language of correctly formatted phone numbers. As before, what we mean are strings that consists of 7 digits, or 3 digits followed by a dash and 4 digits. Figure 4.2 shows the DFA we designed earlier for this language. Recall that in this DFA, the transition label `d` stands for `digit`. For this language, we can give a regular expression that is not only simpler than the DFA but also much more concise:

$$D^7 \cup D^3 - D^4$$

An integer $i$ used as an exponent essentially means "$i$ times".

Compared to the description in words, the regular expression is much more concise since that description used 16 words. In addition, the regular expression allows us to better visualize the strings in the language. □

### Study Questions

4.1.1. What are three frequent advantages of regular expressions over descriptions in words (in a natural language such as English)?

4.1.2. What is a potential advantage of regular expressions over DFA's?

4.1.3. What do a star and an integer exponent mean in a regular expression?

4.1.4. What does the union operator ($\cup$) mean in a regular expression?

### Exercises

4.1.5. Give regular expressions for the languages of the first four exercises of Section 2.2. (Note that $\varepsilon$ can be used in a regular expression.)

## 4.2 Formal Definition

In this section, we formally define what a regular expression is and what a regular expression means, that is, the language that a regular expression describes.

**Definition 4.3** *We say that R is a* regular expression *over alphabet $\Sigma$ if R satisfies one of the following:*

1. $R = a$, *where* $a \in \Sigma$.
2. $R = \varepsilon$.
3. $R = \emptyset$.
4. $R = (R_1 \cup R_2)$, *where each* $R_i$ *is a regular expression over* $\Sigma$.

5. $R = (R_1 \circ R_2)$, *where each* $R_i$ *is a regular expression over* $\Sigma$.

6. $R = (R_1^*)$, *where* $R_1$ *is a regular expression over* $\Sigma$.

Note that, to be precise, these are fully parenthesized regular expressions. But parentheses can be omitted. In that case, we assume that the operations are left associative and that they have the following order of precedence: $*$, then $\circ$, then $\cup$. In addition, the concatenation operator $\circ$ is usually omitted: $R_1R_2$. So, for example,

$$0 \cup 01 \cup 1^* = ((0 \cup (0 \circ 1)) \cup (1^*))$$

and

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

**Definition 4.4** *Suppose that R is a* regular expression *over* $\Sigma$. *The language* described *by R (or the* language *of R) is defined as follows:*

1. $L(a) = \{a\}$, *if* $a \in \Sigma$.

2. $L(\varepsilon) = \{\varepsilon\}$.

3. $L(\emptyset) = \emptyset$.

4. $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$.

5. $L(R_1R_2) = L(R_1)L(R_2)$.

6. $L(R_1^*) = L(R_1)^*$.

These are the basic regular expressions we will use in these notes. When convenient, we will augment them with the following "abbreviations". If $\Sigma =$

$\{a_1, \ldots, a_k\}$ is an alphabet, then $\Sigma$ denotes the regular expression $a_1 \cup \cdots \cup a_k$. If $R$ is a regular expression, then $R^+$ denotes the regular expression $RR^*$, so that

$$\begin{aligned} L(R^+) &= L(R)L(R)^* \\ &= \{x_1 \cdots x_k \mid k \geq 1 \text{ and each } x_i \in L(R)\} \\ &= L(R)^+. \end{aligned}$$

If $R$ is a regular expression and $k$ is a positive integer, then $R^k$ denotes $R$ concatenated with itself $k$ times.

## 4.3 More Examples

In this section, we give several additional examples of regular expressions. In fact, we give regular expressions for almost all the languages of the examples of Section 2.3. The regular expressions will not be necessarily much simpler than DFA's or NFA's for these languages, but they will be more concise.

**Example 4.5** In all of these examples, the alphabet is $\Sigma = \{0, 1\}$.

1. The regular expression $\Sigma^*$ describes the language of all strings over $\Sigma$. In other words, $L(\Sigma^*) = \Sigma^*$.

2. The language of all strings that begin with $1$ is described by the regular expression $1\Sigma^*$.

3. The language of all strings that end in $1$ is described by $\Sigma^*1$.

4. The language of strings of length at least two that begin and end with the same symbol is described by $0\Sigma^*0 \cup 1\Sigma^*1$.

5. The language of strings that contain the substring `001` is described by $\Sigma^*001\Sigma^*$.

6. The language of strings that contain an even number of `1`'s is described by $(0^*10^*1)^*0^*$.

7. The language of strings that contain a number of `1`'s that's a multiple of $k$ is described by $((0^*1)^k)^*0^*$.

□

**Example 4.6** If $R$ is a regular expression, then

$$\begin{aligned} L(R \cup \emptyset) &= L(R) \\ L(R\emptyset) &= \emptyset \\ L(R\varepsilon) &= L(R) \end{aligned}$$

In addition, $L(\varepsilon^*) = \{\varepsilon\}$ and

$$\begin{aligned} L(\emptyset^*) &= \emptyset^* \\ &= \{x_1 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in \emptyset\} \\ &= \{\varepsilon\} \end{aligned}$$

since, as we said earlier, by convention, $x_1 \cdots x_k = \varepsilon$ when $k = 0$. □

## Exercises

4.3.1. Give a regular expression for the language of strings that begin and end with the same symbol. (Note that the strings may have length one.) The alphabet is $\Sigma = \{0, 1\}$.

4.3.2. Give regular expressions for the languages of Exercise 2.3.6.

4.3.3. Give regular expressions for the languages of Exercise 2.3.8.

4.3.4. Give regular expressions for the languages of Exercise 2.3.7.

4.3.5. Give regular expressions for the complement of each of the languages of Exercise 2.3.6.

4.3.6. Give regular expressions for the languages of Exercise 2.5.4.

4.3.7. Give regular expressions for the languages of Exercise 2.5.5.

## 4.4 Converting Regular Expressions into DFA's

In this section, we show that the languages that are described by regular expressions are all regular. And the proof of this result will be constructive: it will provide an algorithm for converting regular expression into NFA's, which can then be converted into DFA's by using the algorithm of the previous chapter. Combined with the fact that it is often easier to write a regular expression than to design a DFA or an NFA, this implies that regular expressions are useful as a tool for obtaining DFA's for many languages.

**Theorem 4.7** *If a language is described by a regular expression, then it is regular.*

**Proof** Suppose that $R$ is a regular expression. We construct an NFA $N$ that recognizes $L(R)$.

The construction is recursive. There are six cases that correspond to the six cases of Definition 4.3. If $R = a$, where $a \in \Sigma$, if $R = \varepsilon$, or if $R = \emptyset$, then $N$ is one of the NFA's shown in Figure 4.3. These are the base cases.


Figure 4.3: NFA's for the languages $\{a\}$, $\{\varepsilon\}$ and $\emptyset$

If $R = R_1 \cup R_2$, if $R = R_1R_2$, or if $R = R_1^*$, then first recursively convert $R_1$ and $R_2$ into $N_1$ and $N_2$ and then combine or transform these NFA's into an NFA $N$ for $L(R)$ by using the constructions we used to prove the closure results of Section 3.4. □

**Example 4.8** Let's convert the regular expression $\mathtt{0}(\mathtt{0} \cup \mathtt{1})^*$ into an NFA. Figures 4.4 and 4.5 show the various steps. Step 1 shows the basic NFA's. Step 2 shows an NFA for $\mathtt{0} \cup \mathtt{1}$. Step 3 shows an NFA for $(\mathtt{0} \cup \mathtt{1})^*$. Figure 4.5 shows the final result, the concatenation of the NFA's from Steps 2 and 3. □

## Exercises

4.4.1. By using the algorithm of this section, convert the following regular expressions into NFA's. In all cases, the alphabet is $\Sigma = \{\mathtt{0}, \mathtt{1}\}$.

a) $\mathtt{0}^* \cup \mathtt{1}^*$.

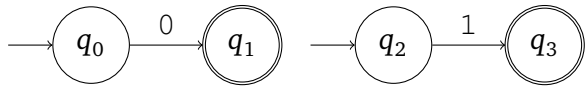
b) $\Sigma^*\mathtt{0}$.

c) $\mathtt{0}^*\mathtt{1}\mathtt{0}^*$.

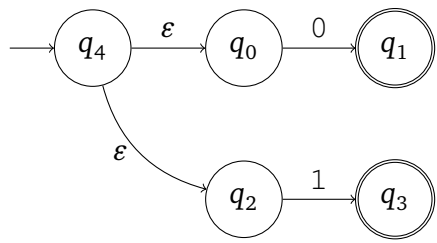
d) $(\mathtt{1}\mathtt{1})^*$.

4.4.2. Extend regular expressions by adding intersection and complementation operators, as in $R_1 \cap R_2$ and $\overline{R_1}$. Revise the formal definitions of Section 4.2

1.




2.




3.

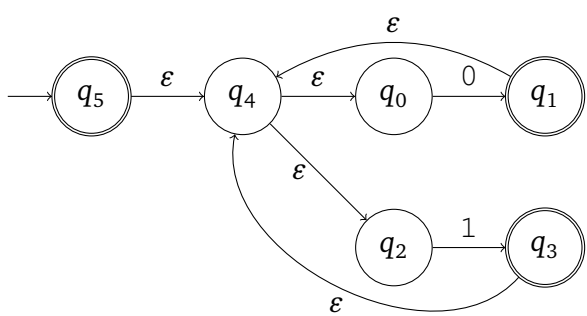



Figure 4.4: Converting $0(0 \cup 1)^*$ into an NFA (part 1)

4.

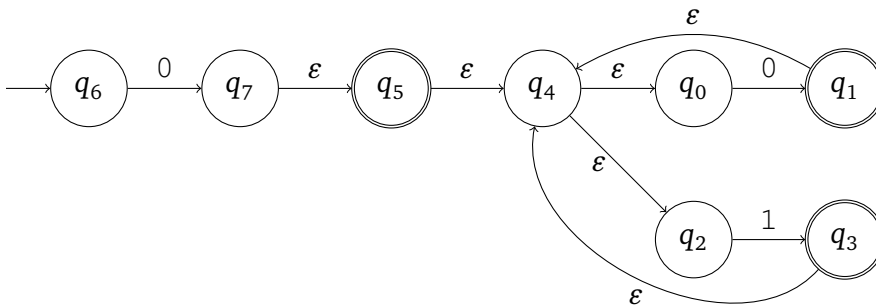



Figure 4.5: Converting $0(0 \cup 1)^*$ into an NFA (part 2)

and then show that these extended regular expressions can only describe regular languages. (In other words, show that these extended regular expressions are no more powerful than the basic ones.)

## 4.5 Converting DFA's into Regular Expressions

In the previous section, we learned that regular expressions can be converted into DFA's. And we said that this was useful because it is often easier to write a regular expression than to design a DFA directly. Compiler-design tools, for example, take advantage of this fact: they take as input regular expressions that specify elements of a programming language and then convert those expressions into DFA's that are eventually turned into code that is incorporated into a compiler. Another example is the standard library of the programming language JavaScript, which allows programmers to specify input string patterns by using

regular expressions.

One question that then arises is whether this strategy of writing a regular expression and then converting it into a DFA applies to every regular language. In other words, are there languages that have DFA's but no regular expression? If so, we would have to design DFA's directly for those languages.

In this section and the next one, we will show that the answer is no: every language that has a DFA also has a regular expression. This shows that the above strategy of writing regular expressions and converting them into DFA's is not more limited than the alternative of designing DFA's directly.

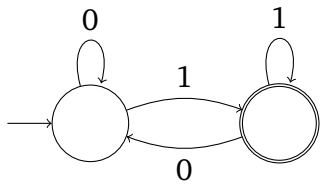
Once again, the proof of this result will be constructive: it will provide an algorithm that converts DFA's into equivalent regular expressions. In principle, this algorithm could be useful. For example, there are languages for which it is easier to design a DFA directly than to write a regular expression. If what we needed for such a language was a regular expression, then it might be easier to design a DFA and convert it into a regular expression than to write a regular expression directly.

So we want an algorithm that converts DFA's into regular expressions. Where should we start? Let's start small. Let's start with the smallest possible DFA's and see how we can convert them into regular expressions.

There are two DFA's that have only one state:

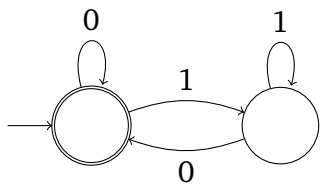

To keep things concrete, we're assuming that the input alphabet $\Sigma$ is $\{0, 1\}$. These DFA's can be easily converted into regular expressions: $\Sigma^*$ and $\emptyset$. This was probably too easy to teach us much about the algorithm.

So let's consider a DFA with two states:




A regular expression for this DFA can be written by allowing the input string to leave and return to the start state any number of times, and then go to the accepting state and stay there. There are two basic ways to leave the start state and return to it: either by reading a 0 or by reading a string of the form $11^*0$. There is only one way of going to the accepting state and staying there: by reading a string of the form $11^*$. So the resulting regular expression is $(0 \cup 11^*0)^*11^*$.

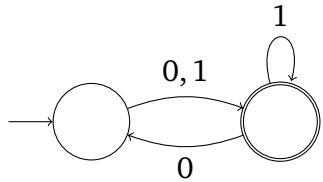
Here's another two-state DFA:




In this case, the regular expression is simpler: $(0 \cup 11^*0)^*$.

It could also be that both states of a two-state DFA are accepting, or that neither state is accepting. Those cases are easy to deal with: $\Sigma^*$ and $\emptyset$.

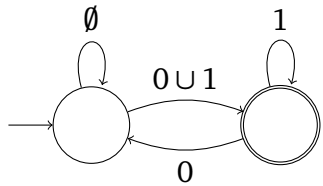
Now, it could be that some of the transitions are missing and that others are labeled by multiple symbols. For example,



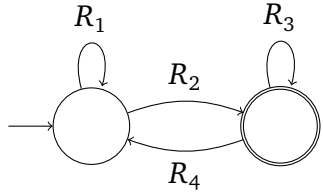

In this case, there is only one way to leave the start state and return to it: by reading a string of the form $(0 \cup 1)1^*0$. This leads to the regular expression $((0 \cup 1)1^*0)^*(0 \cup 1)1^*$.

So transitions labeled with multiple symbols are easy to deal with. But any of the transitions in a two-state DFA can be missing. This would lead to a large number of cases. It would be much more convenient if we could come up with a single "formula" that covers all the cases.

This can be done by considering that missing transitions are actually there but labeled by the regular expression $\emptyset$. This makes sense because the set of symbols that allow us to use a missing transition is empty. In addition, we can consider that transitions labeled with multiple symbols are actually labeled by the union of those symbols. This too makes sense since that's the regular expression that describes the set of symbols that allow us to use that transition. If we transform the previous DFA in this way, we get




This DFA has the following general form:

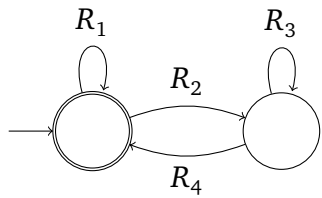



In this DFA, each transition is labeled by a regular expression. That regular expression describes the set of strings that must be read for that transition to be used. Using our earlier reasoning, the set of strings accepted by this general DFA is described by the following regular expression:

$$(R_1 \cup (R_2)(R_3)^*(R_4))^*(R_2)(R_3)^*.$$

It is possible to double-check that the above regular expression is correct by considering all the ways in which any of the transitions could be labeled by the empty set. This would make extensive use of the following basic properties of regular expressions: $R \cup \emptyset = \emptyset \cup R = R$, $R\emptyset = \emptyset R = \emptyset$ and $\emptyset^* = \{\varepsilon\}$.

Another possible general form for a two-state DFA is




Again, using our earlier reasoning, this corresponds to the regular expression

$$(R_1 \cup (R_2)(R_3)^*(R_4))^*.$$

So we now know how to handle any two-state DFA. Let's move on to a DFA with three states, such as the one shown in Figure 4.6. To make things more

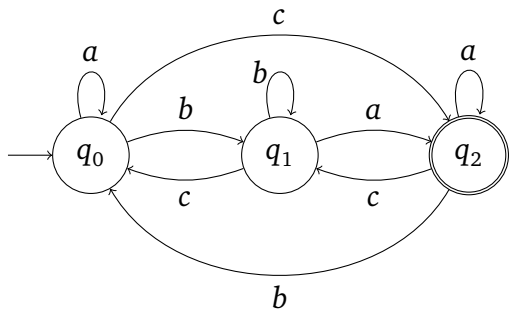



Figure 4.6: A DFA with three states

interesting, the input alphabet is now $\{a, b, c\}$. Coming up with a regular expression for this DFA seems much more complicated than for DFA's with two states. So here's an idea that may at first seem a little crazy: what about we try to remove the middle state of the DFA? If this could be done, then the DFA would have two states and we already know how to handle that.

To be able to remove state $q_1$ from the DFA, we need to consider all the ways in which state $q_1$ can be used to travel through the DFA. For example, state $q_1$ can be used to go from state $q_0$ to state $q_2$. The DFA does that while reading a string of the form $bb^*a$. The eventual removal of state $q_1$ can be compensated for by adding the option $bb^*a$ to the transition that goes directly from $q_0$ to $q_2$, as shown in Figure 4.7. Then, instead of going from $q_0$ to $q_2$ through $q_1$ while reading a string of the form $bb^*a$, we can now go directly from $q_0$ to $q_2$ while reading that same string.

Note that the resulting automaton is no longer a DFA because it now contains a transition that is labeled by a regular expression and that regular expression is not just $\emptyset$ or a union of symbols. But the meaning of this should be pretty clear: if a transition is labeled $R$, then that transition can be used while reading any

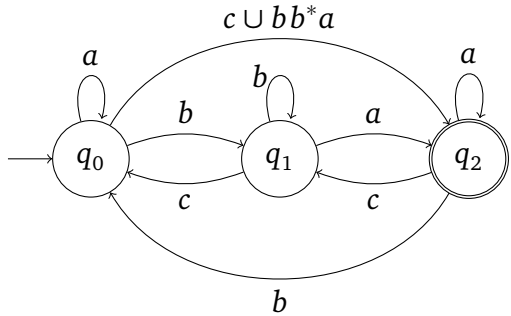



Figure 4.7: Preparing to remove state $q_1$

string in the language described by $R$. In the next section, we will take the time to formally define this new kind of automaton and how it operates.

Another possibility that needs to be considered is that state $q_1$ can also be used to travel from state $q_0$ back to state $q_0$. The DFA does that while reading a string of the form $bb^*c$. Once again, the removal of state $q_1$ can be compensated for by adding to the transition that goes directly from $q_0$ back to $q_0$, as shown in Figure 4.8.

To be able to fully remove $q_1$ from the DFA, this *compensation operation* needs to be performed for every pair of states other than $q_1$. Once this is done and $q_1$ is removed, the result is the two-state automaton shown in Figure 4.9. And we already know how to convert a two-state automaton to a regular expression.

Here's a general description of the compensation operation. Consider the top automaton of Figure 4.10. State $q_1$ is the one we want to remove. That state can be used to travel from $q_2$ to $q_3$ while reading a string of the form $(R_2)(R_3)^*(R_4)$. So we can compensate for the eventual removal of $q_1$ by adding the option $(R_2)(R_3)^*(R_4)$ to the transition going directly from $q_2$ to $q_3$, as shown in the second automaton of Figure 4.10.

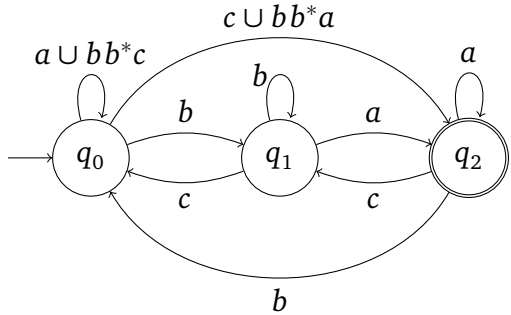



Figure 4.8: Preparing to remove state $q_1$ (continued)

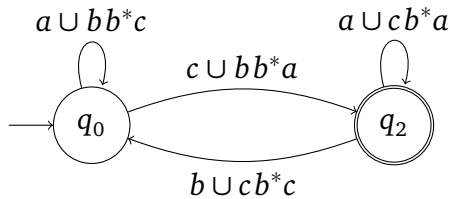



Figure 4.9: The result of removing state $q_1$ from the DFA of Figure 4.6

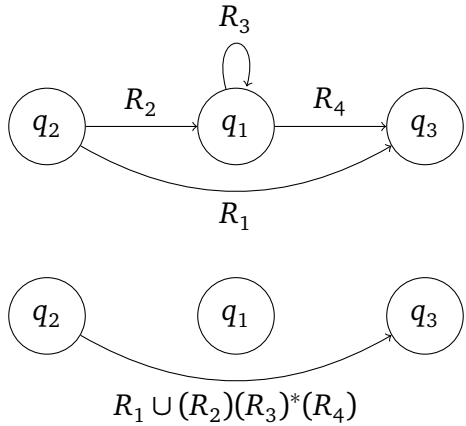



Figure 4.10: The compensation operation

Note that in Figure 4.10, transitions are labeled by regular expressions, not individual symbols. This allows us to deal with missing transitions and transitions labeled with multiple symbols, as explained earlier.

In Figure 4.10, it could also be that $q_2$ and $q_3$ are the same state. In that case, the transition going from $q_2$ to $q_3$ would be a loop.

Now, not all DFA's with three states are similar to the DFA of Figure 4.6. For example, it could be that state $q_1$ is also an accepting state, as shown in Figure 4.11. Removing state $q_1$ would be problematic.

A solution is to modify the DFA so it has only one accepting state. This can be done by adding a new accepting state, new $\varepsilon$ transitions from the old accepting states to the new one, and removing the accepting status of the old accepting states, as shown in Figure 4.12. Once the new accepting state is added, we can remove states $q_1$ and $q_2$, one by one, as before, and end up, once again, with an automaton with two states. The result is shown in Figure 4.13. The first

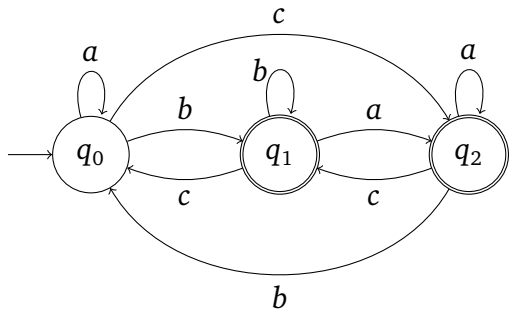



Figure 4.11: A three-state DFA with two accepting states

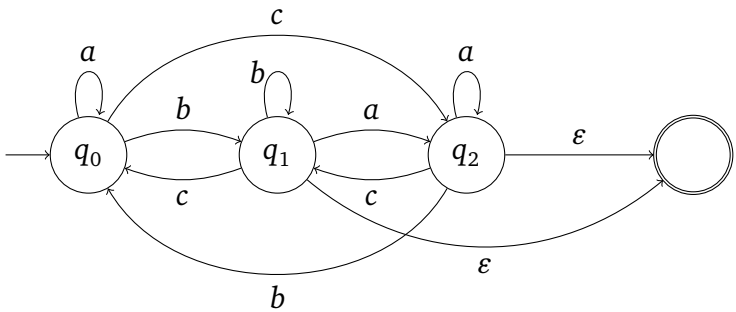
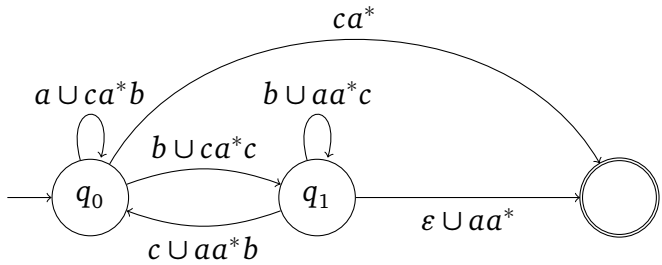



Figure 4.12: Transforming the DFA of Figure 4.11 so it has only one accepting state



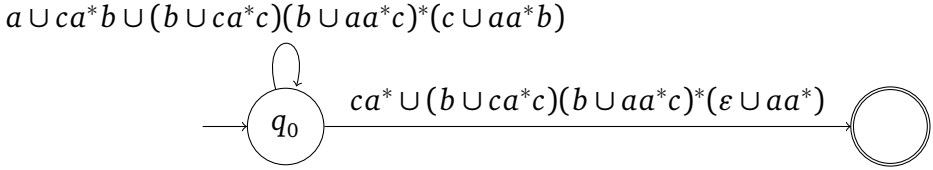
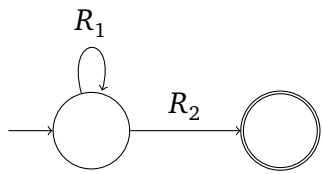




Figure 4.13: Removing states $q_2$ and $q_1$ (in that order) from the automaton of Figure 4.12

automaton is the result of removing $q_2$. The second automaton is the result of removing $q_1$. That automaton is of the form




where

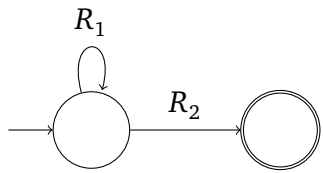
$$R_1 = a \cup ca^*b \cup (b \cup ca^*c)(b \cup aa^*c)^*(c \cup aa^*b)$$

and

$$R_2 = ca^* \cup (b \cup ca^*c)(b \cup aa^*c)^*(\varepsilon \cup aa^*).$$

This automaton is easy to convert into a regular expression: $(R_1)^*(R_2)$.

It's interesting to note that the removal of states $q_1$ and $q_2$ from the automaton of Figure 4.12 resulted in a very simple two-state automaton. This is not an accident: when we add a new accepting state to an automaton in the way we described, the resulting two-state automaton is always of the form




To see why, consider again Figure 4.10, which illustrates the compensation operation. If the new accepting state is involved in a compensation operation, it must be as state $q_3$ because the new accepting state has no outgoing transitions. But then, it's clear that the compensation operation does not add outgoing transitions to that state. Therefore, in the final two-state automaton, the new accepting state will not have any outgoing transitions. Because this results in a simpler two-state automaton, we will always add a new accepting state to the original automaton, even when it only has one accepting state.

In an automaton with three states, it could also be that the start state is an accepting state. But that's not a problem since adding a new accepting state will turn the start state into a non-accepting state so we are back to the previous case.

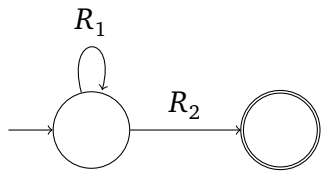
So we now know how to handle any automaton with three states: add a new accepting state and then remove the two states that are not the start state or the new accepting state. This results in a simple two-state automaton that's easy to convert to a regular expression.

What about an automaton with four states? We can simply use the same strategy: add a new accepting state and then remove the three states that are not the start state or the new accepting state. And clearly, this can be extended to automata with any number of states. Here's a complete description of the algorithm:

1. If the automaton has no accepting states, output $\emptyset$.

2. Add a new accepting state, together with new $\varepsilon$ transitions from the old accepting states to the new one. Remove the accepting status of the old accepting states.

3. Remove each state, one by one, except for the start state and the accepting state. This results in an automaton of the following form:



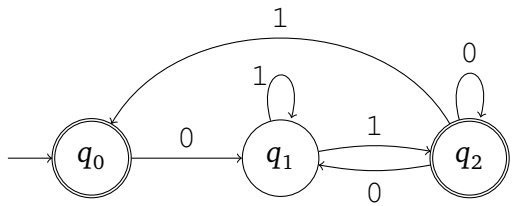

4. Output $(R_1)^*(R_2)$.

This description of the algorithm is fairly informal. We will make it more precise in the next section.

## Exercises

4.5.1. By using the algorithm of this section, convert into regular expressions the NFA's of Figure 4.14.

a)




b)

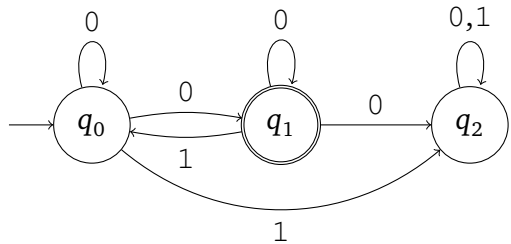



Figure 4.14: NFA's for Exercise 4.5.1

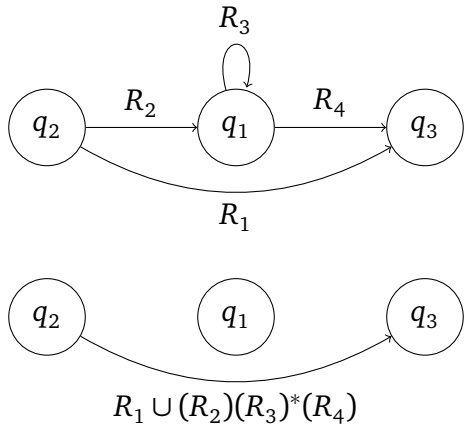



Figure 4.15: The compensation operation

## 4.6 Precise Description of the Algorithm

We now give a precise description of the algorithm of the previous section. Such a description is needed to carefully prove that the algorithm works. Or to code the algorithm.

The key step in the algorithm is the removal of each state other than the start state and the accepting state. This can be done by using the compensation operation we described in the previous section. In general, suppose we want to remove state $q_1$. Let $q_2$ and $q_3$ be two other states and consider the transitions that go from $q_2$ to $q_3$ either directly or through $q_1$, as shown in the first automaton of Figure 4.15. To compensate for the eventual removal of state $q_1$, we add the regular expression $(R_2)(R_3)^*(R_4)$ to the transition that goes directly from $q_2$ to $q_3$, as shown in the second automaton of Figure 4.15.

Note that the description of the compensation operation assumes that there is

exactly one transition going from a state to each other state. This is not the case in every finite automaton since there can be no transitions or multiple transitions going from one state to another. So the first step of the algorithm should actually be to transform the automaton so it has exactly one transition going from each state to every other state. This is easy to do, as explained in the previous section. If there is no transition from $q_1$ to $q_2$, add one labeled $\emptyset$. If there are multiple transitions labeled $a_1, \ldots, a_k$ going from $q_1$ to $q_2$, replace those transitions by a single transition labeled $a_1 \cup \cdots \cup a_k$.

In addition, as mentioned earlier, the compensation operation turns the automaton into one whose transitions are labeled by regular expressions. To be able to describe precisely this operation, and to convince ourselves that it works correctly, we need to define precisely this new type of automaton and how it operates. Here's one way of doing it.

**Definition 4.9** *A* generalized nondeterministic finite automaton (GNFA) *is a* 5*-tuple* $(Q, \Sigma, \delta, q_0, F)$ *where*

1. $Q$ *is a finite set of* states.
2. $\Sigma$ *is an alphabet called the* input alphabet.
3. $\delta : Q \times Q \to \mathcal{R}$ *is the* transition function*, where* $\mathcal{R}$ *is the set of all regular expressions over* $\Sigma$.
4. $q_0 \in Q$ *is the* starting state.
5. $F \subseteq Q$ *is the set of* accepting states.

The only difference between this definition and that of an ordinary NFA is the specification of the transition function. In an NFA, the transition function takes a state and a symbol, or $\varepsilon$, and gives us a set of possible next states. In

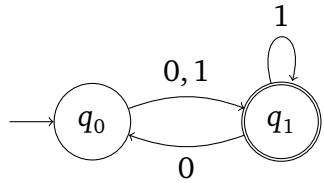



Figure 4.16: An NFA

constrast, in a GNFA, the transition function takes a pair of states and gives us the regular expression that labels the transition going from the first state to the second one. Note how this neatly enforces the fact that in a GNFA, there is exactly one transition going from each state to every other state.

**Example 4.10** Consider the NFA shown in Figure 4.16. This NFA is almost a GNFA. In fact, only two things prevent it from being a GNFA: there is no transition from $q_0$ to $q_0$ and there are two transitions going from $q_0$ to $q_1$ (one labeled 0, the other labeled 1). To turn this NFA into a GNFA, all we need to do is add a transition labeled $\emptyset$ from $q_0$ to $q_0$ and merge the two transitions going from $q_0$ to $q_1$ into a single transition labeled $0 \cup 1$.

Here is a formal description of the resulting GNFA: the GNFA is $(\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\})$ where $\delta$ is defined by the table shown in Figure 4.17. □

We now define how a GNFA operates; that is, we define what it means for a GNFA to accept a string. The idea is that a string $w$ is accepted if the GNFA reads a sequence of strings $y_1, \ldots, y_k$ with the following properties:

1. This sequence of strings takes the GNFA through a sequence of states $r_0, r_1, \ldots, r_k$.

| $\delta$ | $q_0$ | $q_1$ |
|---|---|---|
| $q_0$ | $\emptyset$ | $0 \cup 1$ |
| $q_1$ | $0$ | $1$ |

Figure 4.17: The transition function of the GNFA that corresponds to the NFA of Figure 4.16.

2. This first state $r_0$ is the start state of the GNFA.

3. The last state $r_k$ is an accepting state.

4. The reading of each string $y_i$ is a valid move, in the sense that $y_i$ is in the language of the regular expression that labels the transition going from $r_{i-1}$ to $r_i$.

5. The concatenation of all the $y_i$'s corresponds to $w$.

All of this can be said more concisely, and more precisely, as follows:

**Definition 4.11** *Let $N = (Q, \Sigma, \delta, q_0, F)$ be a GNFA and let $w$ be a string of length $n$ over $\Sigma$. Then $N$* accepts *$w$ if and only if $w$ can be written as $y_1 \cdots y_k$, with each $y_i \in \Sigma^*$, and there is a sequence of states $r_0, r_1, \ldots, r_k$ such that*

$$
\begin{aligned}
&r_0 = q_0 \\
&y_i \in \delta(r_{i-1}, r_i), \quad \text{for } i = 1, \ldots, k \\
&r_k \in F.
\end{aligned}
$$

We now describe precisely the state removal step of the algorithm. Let $q_1$ be

the state to be removed. For every other pair of states $q_2$ and $q_3$, let

$$R_1 = \delta(q_2, q_3)$$
$$R_2 = \delta(q_2, q_1)$$
$$R_3 = \delta(q_1, q_1)$$
$$R_4 = \delta(q_1, q_3)$$

as illustrated by the first automaton of Figure 4.15. Change to $R_1 \cup (R_2)(R_3)^*(R_4)$ the label of the transition that goes directly from $q_2$ to $q_3$, as shown in the second automaton of Figure 4.15. That is, set $\delta(q_2, q_3)$ to $R_1 \cup (R_2)(R_3)^*(R_4)$. Once all pairs $q_2$ and $q_3$ have been considered, remove state $q_1$ and all adjacent transitions (those that come into, or come out of, $q_1$).

And here's a proof that this works:

**Lemma 4.12** *If the state removal step is applied to GNFA N, then the resulting GNFA still recognizes $L(N)$.*

**Proof** Let $N'$ be the GNFA that results from removing state $q_1$ from GNFA $N$. Suppose that $w \in L(N)$. If $w$ can be accepted by $N$ without traveling through state $q_1$, then $w$ is accepted by $N'$. Now suppose that to accept $w$, $N$ must travel through $q_1$. Suppose that in one such instance, $N$ reaches $q_1$ from $q_2$ and that it goes to $q_3$ after leaving $q_1$, as shown in the first automaton of Figure 4.15. Then $N$ travels from $q_2$ to $q_3$ by reading strings $y_1, \ldots, y_k$ such that

$$\begin{aligned} &y_1 \in L(R_2) \\ &y_i \in L(R_3), \quad \text{for } i = 2, \ldots, k-1 \\ &y_k \in L(R_4). \end{aligned}$$

This implies that $y_1 \cdots y_k \in L((R_2)(R_3)^*(R_4))$. In $N'$, the transition going directly

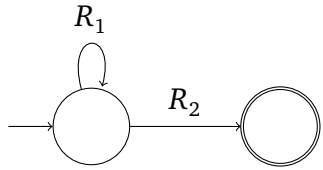
from $q_2$ to $q_3$ is now labeled $R_1 \cup (R_2)(R_3)^*(R_4)$. This implies that $N'$ can move directly from $q_2$ to $q_3$ by reading $y_1 \cdots y_k$. And this applies to every instance in which $N'$ goes through $q_1$ while reading $w$. Therefore, $N'$ still accepts $w$.

Now suppose that $w \in L(N')$. If $w$ can be accepted by $N'$ without using one of the relabeled transitions, then $w$ is accepted by $N$. Now suppose that to accept $w$, $N'$ must use one of the relabeled transitions. Suppose that in one such instance, $N'$ travels from $q_2$ to $q_3$ on a transition labeled $R_1 \cup (R_2)(R_3)^*(R_4)$, as shown in the second automaton of Figure 4.15. If $N'$ goes from $q_2$ to $q_3$ by reading a string $y \in L(R_1)$, then $N$ could have done the same. If instead $N'$ goes from $q_2$ to $q_3$ by reading a string $y \in L((R_2)(R_3)^*(R_4))$, then it must be that $y = y_1 \cdots y_k$ with

$$\begin{aligned} &y_1 \in L(R_2) \\ &y_i \in L(R_3), \quad \text{for } i = 2, \ldots, k-1 \\ &y_k \in L(R_4). \end{aligned}$$

This implies that $N$ can go from $q_2$ to $q_3$ while reading $y$ as long as it does it by going through $q_1$ while reading $y_1, \ldots, y_k$. This applies to every instance in which $N'$ uses a relabeled transition while reading $w$. Therefore, $N$ accepts $w$ and this completes the proof that $L(N') = L(N)$. □

So we now know for sure that the state removal step works. As described in the previous section, this step is repeated for every state except for the start state and the new accepting state. As explained in the previous section, this always results in a two-state GNFA of the following form:




We are now ready to describe the entire conversion algorithm and prove its correctness. The algorithm can actually convert arbitrary NFA's, not just DFA's.

**Theorem 4.13** *If a language is regular, then it can be described by a regular expression.*

**Proof** Suppose that $L$ is a regular language and that NFA $N$ recognizes $L$. We construct a regular expression $R$ that describes $L$ by using the following algorithm:

1. If $N$ has no accepting states, set $R = \emptyset$ and stop.

2. Transform $N$ into a GNFA. This can be done as follows. For every pair of states $q_1$ and $q_2$, if there is no transition from $q_1$ to $q_2$, add one labeled $\emptyset$. If there are multiple transitions labeled $a_1, \ldots, a_k$ going from $q_1$ to $q_2$, replace those transitions by a single transition labeled $a_1 \cup \cdots \cup a_k$.

3. Add a new accepting state to $N$, add new $\varepsilon$ transitions from the old accepting states to the new one and remove the accepting status of the old accepting states.

4. One by one, remove every state other than the start state or the accepting state. This can be done as follows. Let $q_1$ be the state to be removed. For

every other pair of states $q_2$ and $q_3$, let

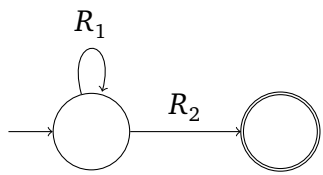
$$R_1 = \delta(q_2, q_3)$$
$$R_2 = \delta(q_2, q_1)$$
$$R_3 = \delta(q_1, q_1)$$
$$R_4 = \delta(q_1, q_3)$$

as illustrated by the first automaton of Figure 4.15. Change to $R_1 \cup (R_2)(R_3)^*(R_4)$ the label of the transition that goes directly from $q_2$ to $q_3$, as shown in the second automaton of Figure 4.15. That is, set

$$\delta(q_2, q_3) = R_1 \cup (R_2)(R_3)^*(R_4).$$

Once all pairs $q_2$ and $q_3$ have been considered, remove state $q_1$.

5. The resulting GNFA is of the form




Set $R = (R_1)^*(R_2)$.

It is not hard to see that this algorithm is correct, mainly because each transformation of $N$ preserves the fact that $N$ recognizes $L$. In the case of the state removal step, this was established by the lemma. □

## Exercises

4.6.1. Give a formal description of the GNFA of Figure 4.8.

4.6.2. Give a formal description of the GNFA of Figure 4.9.

# Chapter 5

# Nonregular Languages

In this chapter, we show that not all languages are regular. In other words, we show that there are languages that can't be recognized by finite automata or described by regular expressions. We will learn to prove that particular languages are nonregular by using a general result called the Pumping Lemma.

## 5.1 Some Examples

So far in these notes, we have seen many examples of regular languages. In this section, we discover our first examples of nonregular languages. In all cases, the alphabet is $\{0, 1\}$.

**Example 5.1** Consider the language of strings of the form $0^n1^n$, where $n \geq 0$. Call it $L$. We know that languages such as $\{0^i1^j \mid i, j \geq 2\}$ and $\{0^i1^j \mid i \equiv j \pmod 3\}$ are regular. But here we want to determine if $i = j$. One strategy a DFA could attempt to use is to count the number of 0's, count the number of 1's and then verify that the two numbers are the same.

As we have seen before, one way that a DFA can count 0's is to move to a different state every time a 0 is read. For example, if the initial state of the DFA is $q_0$, then the DFA could go through states $q_0, q_1, q_2, \ldots, q_n$ as it reads $n$ 0's. Counting from 0 to $n$ requires $n+1$ different states, and $n$ can be arbitrary large. But a DFA only has a finite number of states! If $n$ is greater than or equal to that number, then the DFA doesn't have enough states to implement this strategy.

This argument shows that this particular strategy of counting the number of 0's by using different states is not going to lead to a DFA that recognizes the language $L$. It's hard to imagine how a DFA could recognize $L$ without counting the number of 0's. But because we can't think of a way doesn't mean that one doesn't exist. To *prove* that $L$ is not regular, we need a more general argument, one that shows that no DFA, using any strategy whatsoever, can recognize this language.

Such an argument can be constructed by considering the states that a DFA goes through as it reads the 0's of a string of the form $0^n1^n$. Not every DFA counts 0's but every DFA goes through states as it reads the input.

So suppose that $M$ recognizes $L$. Consider a string $w$ of the form $0^n1^n$. As $M$ reads the 0's of $w$, it goes through a sequence of states $r_0, r_1, r_2, \ldots, r_n$. Choose $n$ to be equal to the number of states of $M$. Then there must be a repetition in the sequence.

Suppose that $r_i = r_j$ with $i < j$. Then the computation of $M$ on $w$ is as shown in Figure 5.1. That is, after reading $i$ 0's, the DFA enters state $r_i$. After reading an additional $j-i$ 0's, for a total of $j$ 0's, the DFA returns to state $r_i$. From there, after reading the rest of $w$, which equals $0^{n-j}1^n$, the DFA reaches an accepting state.[1]

[1]Strictly speaking, when reading $0^{n-j}1^n$ from state $r_i$, $M$ will actually continue in the loop until the first 1 is read. The diagram of Figure 5.1 is only meant to illustrate the fact that after reading $0^{n-j}1^n$ from $r_i$, the DFA *eventually* reaches an accepting state.

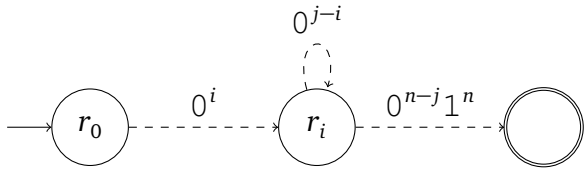



Figure 5.1: The computation of $M$ on the string $0^n1^n$

Now, what Figure 5.1 makes clear is that $M$ accepts not only $w$ but also the string $0^i0^{n-j}1^n$, which is simply $w$ with $j-i$ 0's removed. This string is accepted by $M$ because after reading $0^i$, the DFA enters state $r_i$. From there, after reading $0^{n-j}1^n$, the DFA again reaches the same accepting state that was reached when reading $w$. In other words, when reading this string, $M$ simply skips the loop shown in Figure 5.1.

But $0^i0^{n-j}1^n$ is not in $L$ because it contains less 0's than 1's. So we have found a string that is accepted by $M$ but does not belong to $L$. This contradicts the fact that $M$ recognizes $L$ and shows that a DFA that recognizes $L$ cannot exist. Therefore, $L$ is not regular. □

The argument we developed in this example can be used to show that other languages are not regular.

**Example 5.2** Let $L$ be the language of strings of the form $0^i1^j$ where $i \leq j$. Once again, suppose that $L$ is regular. Let $M$ be a DFA that recognizes $L$ and let $n$ be the number of states of $M$.

Consider the string $w = 0^n1^n$. As $M$ reads the 0's in $w$, $M$ goes through a sequence of states $r_0, r_1, r_2, \ldots, r_n$. Because this sequence is of length $n+1$, there must be a repetition in the sequence.

Suppose that $r_i = r_j$ with $i < j$.[2] Then the computation of $M$ on $w$ is once again as shown in Figure 5.1. And, once again, this implies that the string $0^i 0^{n-j} 1^n$ is also accepted. But since the number of 0's has been reduced, this string happens to be in $L$. So there is no contradiction...

So by skipping the loop, we get a string that is still in the language. Let's try something else: let's go around the loop twice. This corresponds to reading the string $0^i 0^{j-i} 0^{j-i} 0^{n-j} 1^n = 0^{n+(j-i)} 1^n$. This string is also accepted by $M$. However, since this string has more 0's than 1's, it is not in $L$. This contradicts the fact that $M$ recognizes $L$. Therefore, $M$ cannot exist and $L$ is not regular. □

What this last example shows is that sometimes going around the loop more than once is the way to obtain a contradiction and make the argument work.

Here's another example.

**Example 5.3** Let $L$ be the language of strings of the form $ww$ where $w$ is any string over the alphabet is $\{0, 1\}$. Suppose that $L$ is regular. Let $M$ be a DFA that recognizes $L$ and let $n$ be the number of states of $M$.

Consider the string $w = 0^n 0^n$. As $M$ reads the 0's in the first half of $w$, $M$ goes through a sequence of states $r_0, r_1, r_2, \ldots, r_n$. Because this sequence is of length $n + 1$, there must be a repetition in the sequence.

Suppose that $r_i = r_j$ with $i < j$. Then the computation of $M$ on $w$ is as shown in Figure 5.2. This implies that the string $0^{2n-(j-i)}$ is also accepted. Unfortunately, we cannot conclude that this string is not in $L$. In fact, if $j - i$ is even, then this string *is* in $L$ because the total number of 0's would still be even.

[2]Note that these $i$ and $j$ are not the same $i$ and $j$ that were used to define the language. It may have been clearer use different names in the definition of the language, for example, by saying that $L$ is the language of strings of the form $0^s 1^t$ where $s \leq t$. But at some point, you start running out of convenient names. So you can just view the $i$ and $j$ in the definition of $L$ as being *local* to that definition, just as in programming, variables can be local to a function, which allows you to reuse their names for other purposes outside of that function.

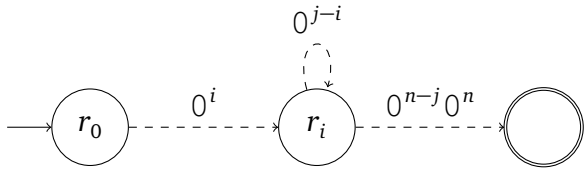



Figure 5.2: The computation of $M$ on the string $0^n0^n$

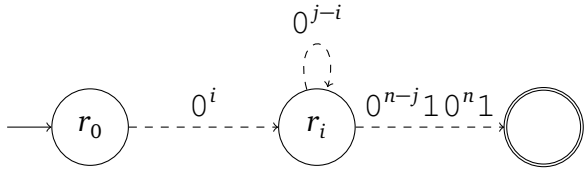



Figure 5.3: The computation of $M$ on the string $0^n10^n1$

To fix this argument, note that if the string is still in $L$ after we delete some $0$'s from its first half, it's because the middle of the string will have shifted to the right. So we just need to pick a string $w$ whose middle cannot move...

Let $w = 0^n10^n1$. As before, we get a repetition within the first block of $0$'s, as shown in Figure 5.3. This implies that the string $0^{n-(j-i)}10^n1$ is also accepted by $M$. If this string was in $L$, since its second half ends in $1$, its first half would also have to end in $1$. But then, the two halves would not be equal. So this string cannot be in $L$, which contradicts the fact that $M$ recognizes $L$. Therefore, $M$ does not exist and $L$ is not regular. □

This last example makes the important point that the string $w$, that is, the string that we focus the argument on, sometimes has to be chosen carefully. In fact, this choice can sometimes be a little tricky.

### Exercises

5.1.1. Show that the language $\{0^n 1^{2n} \mid n \geq 0\}$ is not regular.

5.1.2. Show that the language $\{0^i 1^j \mid 0 \leq i \leq 2j\}$ is not regular.

5.1.3. If $w$ is a string, let $w^{\mathcal{R}}$ denote the *reverse* of $w$. That is, if $w = w_1 \cdots w_n$, then $w^{\mathcal{R}} = w_n \cdots w_1$. Show that the language $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$ is not regular.

## 5.2 The Pumping Lemma

In the previous section, we showed that three different languages are not regular. Each example involved its own particular details but all three arguments followed the same pattern and had a lot of details in common. In this section, we will isolate the common portion of these arguments and turn in into the proof of a general result about regular languages. In this way, the common details won't need to be repeated each time we want to show that a language is not regular.

In all three examples, the proof that $L$ is not regular went as follows. Suppose that $L$ is regular. Let $M$ be a DFA that recognizes $L$ and let $n$ be the number of states of $M$.

Choose a string $w \in L$ of length at least $n$. (The choice of $w$ depends on $L$.) As $M$ reads the first $n$ symbols of $w$, $M$ goes through a sequence of states $r_0, r_1, r_2, \ldots, r_n$. Because this sequence is of length $n+1$, there must be a repetition in the sequence.

Suppose that $r_i = r_j$ with $i < j$. Then the computation of $M$ on $w$ is as shown in Figure 5.4. More precisely, if $w = w_1 \cdots w_m$, then $x = w_1 \cdots w_i$, $y = w_{i+1} \cdots w_j$ and $z = w_{j+1} \cdots w_m$.

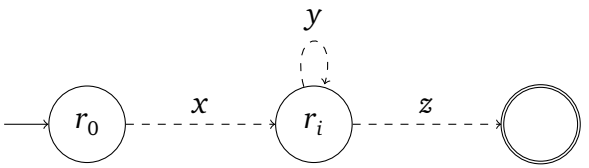



Figure 5.4: The computation of $M$ on the string $w = 0^k1^k$

The fact that the reading of $y$ takes $M$ from state $r_i$ back to state $r_i$ implies that the strings $xz$ and $xy^2z$ are also accepted by $M$. (In fact, it is also true that $xy^kz$ is accepted by $M$ for every $k \geq 0$.)

In each of the examples of the previous section, we then observed that one of the strings $xz$ or $xy^2z$ does not belong to $L$. (The details depend on $x$, $y$, $z$ and $L$.) This contradicts the fact that $M$ recognizes $L$. Therefore, $M$ cannot exist and $L$ is not regular.

To summarize, the argument starts with a language $L$ that is assumed to be regular and a string $w \in L$ that is long enough. It then proceeds to show that $w$ can be broken into three pieces, $w = xyz$, such that $xy^kz \in L$, for every $k \geq 0$.

There is more to it, though. In all three examples of the previous section, we also used the fact that the string $y$ occurs within the first $n$ symbols of $w$. This is because a repetition is guaranteed to occur while $M$ is reading the first $n$ symbols of $w$. This is useful because it gives us some information about the contents of $y$. For example, in the case of the language $\{0^n1^n\}$, this told us that $y$ contained only 0's, which meant that $xz$ contained less 0's than 1's.

In addition, the above also implicitly uses the fact that $y$ is not empty. And this is absolutely critical since, otherwise, $xz$ would equal $w$ and the fact that $xz$ is accepted by $M$ could not possibly lead to a contradiction.

So a more complete summary of the above argument is as follows. The argu-

ment starts with a language $L$ that is assumed to be regular and a string $w \in L$ that is long enough. It then proceeds to show that $w$ can be broken into three pieces, $w = xyz$, such that

1. The string $y$ can be *pumped*, in the sense that $xy^kz \in L$, for every $k \geq 0$.
2. The string $y$ occurs towards the beginning of $w$.
3. The string $y$ is nonempty.

In other words, the above argument proves the following result, which is called the *Pumping Lemma*:

**Theorem 5.4 (Pumping Lemma)** *If $L$ is a regular language, then there is a number $p > 0$, called the* pumping length*, such that if $w \in L$ and $|w| \geq p$, then $w$ can be written as $xyz$ where*

1. *$|xy| \leq p$,*
2. *$y \neq \varepsilon$,*
3. *$xy^kz \in L$, for every $k \geq 0$.*

In a moment, we are going to see how the Pumping Lemma can be used to simplify proofs that languages are not regular. But first, here's a clean write-up of the proof of the Pumping Lemma.

**Proof** Let $L$ be a regular and let $M$ be a DFA that recognizes $L$. Let $p$ be the number of states of $M$. Now, suppose that $w$ is any string in $L$ with $|w| \geq p$. As $M$ reads the first $p$ symbols of $w$, $M$ goes through a sequence of states $r_0, r_1, r_2, \ldots, r_p$. Because this sequence is of length $p + 1$, there must be a repetition in the sequence.

Suppose that $r_i = r_j$ with $i < j$. Then the computation of $M$ on $w$ is as shown in Figure 5.4. In other words, if $w = w_1 \cdots w_m$, let $x = w_1 \cdots w_i$, $y = w_{i+1} \cdots w_j$ and $z = w_{j+1} \cdots w_m$. Clearly, $w = xyz$. In addition, $|xy| = j \leq p$ and $|y| = j - i > 0$, which implies that $y \neq \varepsilon$.

Finally, the fact that the reading of $y$ takes $M$ from state $r_i$ back to state $r_i$ implies that the string $xy^kz$ is accepted by $M$, and thus belongs to $L$, for every $k \geq 0$. □

We now see how the Pumping Lemma can be used to prove that languages are not regular.

**Example 5.5** Let $L$ be the language $\{0^n1^n \mid n \geq 0\}$. Suppose that $L$ is regular. Let $p$ be the pumping length given by the Pumping Lemma. Consider the string $w = 0^p1^p$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, $w$ can be written as $xyz$ where

1. $|xy| \leq p$.

2. $y \neq \varepsilon$.

3. $xy^kz \in L$, for every $k \geq 0$.

Condition (1) implies that $y$ contains only 0's. Condition (2) implies that $y$ contains at least one 0. Therefore, the string $xz$ cannot belong to $L$ because it contains less 0's than 1's. This contradicts Condition (3). So it must be that our initial assumption was wrong: $L$ is not regular. □

It is interesting to compare this proof that $\{0^n1^n\}$ is not regular with the proof we gave in the first example of the previous section. The new proof is shorter

but, perhaps more importantly, it doesn't need to establish that the string $y$ can be pumped. Those details are now hidden in the proof of the Pumping Lemma.[3]

Here's another example.

**Example 5.6** Let $L$ be the language of strings of the form $ww$. Suppose that $L$ is regular. Let $p$ be the pumping length given by the Pumping Lemma. Consider the string $w = 0^p10^p1$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, $w$ can be written as $xyz$ where

1. $|xy| \leq p$.
2. $y \neq \varepsilon$.
3. $xy^kz \in L$, for every $k \geq 0$.

Condition (1) implies that $y$ contains only 0's from the first half of $w$. Condition (2) implies that $y$ contains at least one 0. Therefore, $xz$ is of the form $0^i10^p1$ where $i < p$. This string is not in $L$, contradicting Condition (3). This implies that $L$ is not regular. □

Here's an example for a language of a different flavor.

**Example 5.7** Let $L$ be the language $\{1^{n^2} \mid n \geq 0\}$. That is, $L$ consists of strings of 1's whose length is a perfect square. Suppose that $L$ is regular. Let $p$ be the pumping length. Consider the string $w = 1^{p^2}$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, $w$ can be written as $xyz$ where

1. $|xy| \leq p$.
2. $y \neq \varepsilon$.

[3] Yes, this is very similar to the idea of abstraction in software design.

3. $xy^kz \in L$, for every $k \geq 0$.

Consider the string $xy^2z$, which is equal to $1^{p^2+|y|}$. Since $|y| \geq 1$, for this string to belong to $L$, it must be that $p^2 + |y| \geq (p+1)^2$, the first perfect square greater than $p^2$. But $|y| \leq p$ implies that $p^2 + |y| \leq p^2 + p = p(p+1) < (p+1)^2$. Therefore, $xy^2z$ does not belong to $L$, which contradicts the Pumping Lemma and shows that $L$ is not regular. □

Here's an example that shows that the Pumping Lemma must be used carefully.

**Example 5.8** Let $L$ be the language $\{1^n \mid n \text{ is even}\}$. Suppose that $L$ is regular. Let $p$ be the pumping length. Consider the string $w = 1^{2p}$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, $w$ can be written as $xyz$ where

1. $|xy| \leq p$.

2. $y \neq \varepsilon$.

3. $xy^kz \in L$, for every $k \geq 0$.

Let $y = 1$. Then $xz = 1^{2p-1}$ is of odd length and cannot belong to $L$. This contradicts the Pumping Lemma and shows that $L$ is not regular.

Of course, this doesn't make sense because we know that $L$ *is* regular. So what did we do wrong in this "proof"? What we did wrong is that we *chose* the value of $y$. We can't do that. All that we know about $y$ is what the Pumping Lemma says: $w = xyz$, $|xy| \leq p$, $y \neq \varepsilon$ and $xy^kz \in L$, for every $k \geq 0$. This does not imply that $y = 1$.

To summarize, when using the Pumping Lemma to prove that a language is not regular, we are free to choose the string $w$ and the number $k$. But we cannot choose the number $p$ or the strings $x$, $y$ and $z$. □

One final example.

**Example 5.9** Let $L$ be the language of strings that contain an equal number of $\mathtt{0}$'s and $\mathtt{1}$'s. We could show that this language is nonregular by using the same argument we used for $\{\mathtt{0}^n\mathtt{1}^n\}$. However, there is a simpler proof. The key is to note that

$$L \cap \mathtt{0}^*\mathtt{1}^* = \{\mathtt{0}^n\mathtt{1}^n\}.$$

If $L$ was regular, then $\{\mathtt{0}^n\mathtt{1}^n\}$ would also be regular because the intersection of two regular languages is always regular. But $\{\mathtt{0}^n\mathtt{1}^n\}$ is not regular, which implies that $L$ cannot be regular. □

This example shows that closure properties can also be used to show that languages are not regular. In this case, the fact that $L$ is nonregular became a direct consequence of the fact that $\{\mathtt{0}^n\mathtt{1}^n\}$ is nonregular.

Note that it is not correct to say that the fact that $\{\mathtt{0}^n\mathtt{1}^n\}$ is a subset of $L$ implies that $L$ is nonregular. Because the same argument would also imply that $\{\mathtt{0}, \mathtt{1}\}^*$ is nonregular. In other words, the fact that a language contains a nonregular language does not imply that the larger language is nonregular. What makes a language is nonregular is not the fact that it contains certain strings: it's the fact that it contains certain strings while excluding certain others.

## Exercises

5.2.1. Use the Pumping Lemma to show that the language $\{\mathtt{0}^i\mathtt{1}^j \mid i \leq j\}$ is not regular.

5.2.2. Use the Pumping Lemma to show that the language $\{\mathtt{1}^i\#\mathtt{1}^j\#\mathtt{1}^{i+j}\}$ is not regular. The alphabet is $\{\mathtt{1}, \#\}$.

5.2.3. Exercise 2.4.8 asked you to show that DFA's can add when the numbers are presented in a particular way. In contrast, this exercise asks you to show that DFA's cannot add when the numbers are presented in a more usual way. That is, consider the language of strings of the form $x\#y\#z$ where $x$, $y$ and $z$ are strings of digits that, when viewed as numbers, satisfy the equation $x + y = z$. For example, the string `123#47#170` is in this language because $123 + 47 = 170$. The alphabet is $\{0, 1, \ldots, 9, \#\}$. Use the Pumping Lemma to show that this language is not regular.

5.2.4. Let $L$ be the language of strings that start with `0`. What is wrong with the following "proof" that $L$ is not regular?

> Suppose that $L$ is regular. Let $p$ be the pumping length. Suppose that $p = 1$. Consider the string $w = 01^p$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, $w$ can be written as $xyz$ where
>
> 1. $|xy| \leq p$.
> 2. $y \neq \varepsilon$.
> 3. $xy^kz \in L$, for every $k \geq 0$.
>
> Since $|xy| \leq p = 1$ and since $y \neq \varepsilon$, it must be that $x = \varepsilon$ and that $y = 0$. Therefore, $xz = 1^p$ and this string is not in $L$. This contradicts the Pumping Lemma and shows that $L$ is not regular.

# Chapter 6

# Context-Free Languages

In this chapter, we move beyond regular languages. We will learn an extension of regular expressions called context-free grammars. We will see several examples of languages that can be described by context-free grammars. We will learn to prove that some languages cannot be described even by context-free grammars. We will also discuss algorithms for context-free languages.

## 6.1 Introduction

We know that the language of strings of the form $0^n1^n$ is not regular. This implies that no regular expression can describe this language. But here is a way to describe this language:

$$S \to 0S1$$
$$S \to \varepsilon$$

This is called a *context-free grammar* (CFG). A CFG consists of *variables* and *rules*. This grammar has one variable: $S$. That variable is also the *start variable* of the grammar, as indicated by its placement on the left of the first rule.

Each rule specifies that the variable on the left can be replaced by the string on the right. A grammar is used to *derive* strings. This is done by beginning with the start variable and then repeatedly applying rules until all the variables are gone. For example, this grammar can derive the string `0011` as follows:

$$S \Rightarrow \texttt{0}S\texttt{1} \Rightarrow \texttt{00}S\texttt{11} \Rightarrow \texttt{0011}.$$

The above is called a *derivation*. It is a sequence of steps. In each step, a single rule is applied to one of the variables. For example, the first two steps in this derivation are applications of the first rule of the grammar. The last step is an application of the second rule.

The language *generated* by a grammar, or the language *of* the grammar, is the set of strings it can derive. It's is not hard to see that the above grammar generates the language $\{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$.

Here is another example of a CFG. Consider the language of valid C++ identifiers. Recall that these are strings that begin with an underscore or a letter followed by any number of underscores, letters and digits. We can think of rules in a mechanistic way, as specifying what can be done with a variable. But it is often better to think of variables as representing concepts and of rules as defining the meaning of those concepts.

For example, let $I$ represent a valid identifier. Then the rule $I \to FR$ says that an identifier consists of a first character (represented by $F$) followed by the rest of the identifier ($R$). The rule $F \to \_ \mid L$ says that the first character is either an underscore or a letter. The vertical bar (|) can be viewed as an *or* operator or as a way to combine multiple rules into one. For example, the above rule is

equivalent to the two rules

$$F \to \_$$
$$F \to L$$

Continuing in this way, we get the following grammar for the language of valid identifiers:

$$I \to FR$$
$$F \to \_ \mid L$$
$$R \to \_R \mid LR \mid DR \mid \varepsilon$$
$$L \to \texttt{a} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \cdots \mid \texttt{Z}$$
$$D \to \texttt{0} \mid \cdots \mid \texttt{9}$$

The most interesting rules in this grammar are probably the rules for the variable $R$. These rules define the concept $R$ in a recursive way. They illustrate how repetition can be carried out in a CFG.

## Study Questions

6.1.1. What does a CFG consist of?

6.1.2. What is it that appears on the left-hand-side of a rule? On the right-hand-side?

6.1.3. What is a derivation?

6.1.4. What is the purpose of the vertical bar when it appears on the right-hand-side of a rule?

### Exercises

6.1.5. Consider the grammar for valid identifiers we saw in this section. Show how the strings `x23` and `_v2_` can be derived by this grammar. In each case, give a derivation.

6.1.6. Give CFG's for the languages of the first three exercises of Section 2.2.

## 6.2 Formal Definition of CFG's

In this section, we define formally what a context-free grammar is and the language generated by a context-free grammar.

**Definition 6.1** *A* context-free grammar (CFG) *$G$ is a 4-tuple $(V, \Sigma, R, S)$ where*

1. *$V$ is a finite set of* variables.
2. *$\Sigma$ is a finite set of* terminals.
3. *$R$ is a finite set of* rules *of the form $A \to w$ where $A \in V$ and $w \in (V \cup \Sigma)^*$.*
4. *$S \in V$ is the* start variable.

*It is assumed that $V$ and $\Sigma$ are disjoint.*

**Definition 6.2** *Suppose that $G = (V, \Sigma, R, S)$ is a CFG. Consider a string $uAv$ where $A \in V$ and $u, v \in (V \cup \Sigma)^*$. If $A \to w$ is a rule, then we say that the string $uwv$ can be* derived (in one step) *from $uAv$ and we write $uAv \Rightarrow uwv$.*

**Definition 6.3** *Suppose that $G = (V, \Sigma, R, S)$ is a CFG. If $u, v \in (V \cup \Sigma)^*$, then $v$ can be* derived *from $u$, or $G$* derives *$v$ from $u$, if $v = u$ or if there is a sequence $u_1, u_2, \ldots, u_k \in (V \cup \Sigma)^*$, for some $k \geq 0$, such that*

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v.$$

*In this case, we write $u \overset{*}{\Rightarrow} v$.*

**Definition 6.4** *Suppose that $G = (V, \Sigma, R, S)$ is a CFG. Then the language* generated *by $G$ (or the language* of *$G$) is the following set:*

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

**Definition 6.5** *A language is* context-free *if it is generated by some CFG.*

## 6.3 More Examples

In this section, we give some additional examples of CFG's. In the first four examples, the alphabet is $\{0, 1\}$.

**Example 6.6** The language of all strings is generated by the following grammar:

$$S \to 0S \mid 1S \mid \varepsilon$$

In this grammar, strings are derived from left to right, and then $S$ is deleted. For example,

$$S \Rightarrow 0S \Rightarrow 01S \Rightarrow 011S \Rightarrow 011$$

In contrast, recall the grammar for the language of strings of the form $0^n1^n$:

$$S \to 0S1 \mid \varepsilon$$

In this grammar, strings are derived from the outside towards the middle:

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111.$$

□

**Example 6.7** The language of strings that start with 1:

$$S \to 1T$$
$$T \to 0T \mid 1T \mid \varepsilon$$

The rule for $S$ must be used first and this rule ensures that the derived string begins with a 1.

Here's an alternative grammar for this language:

$$S \to S0 \mid S1 \mid 1$$

The grammar in the previous example derived strings from left to right and then deleted $S$. What this new grammar does is derive strings from right to left and then replace $S$, which is now at the front, by 1. For example,

$$S \Rightarrow S0 \Rightarrow S10 \Rightarrow S010 \Rightarrow 1010$$

□

**Example 6.8** The language of strings that start and end with the same symbol:

$$S \to 0T0 \mid 1T1 \mid 0 \mid 1$$
$$T \to 0T \mid 1T \mid \varepsilon$$

Note that the following grammar would not work:

$$S \to XTX \mid 0 \mid 1$$
$$X \to 0 \mid 1$$
$$T \to 0T \mid 1T \mid \varepsilon$$

In this grammar, to derive a string of length greater than 1, we'd have to use the first rule:

$$S \Rightarrow XTX$$

But then, there is no way to enforce that the same symbols will be derived from the two occurrences of $X$. So this won't guarantee that the string begins and ends with the same symbol. □

**Example 6.9** The language of strings that contain the substring `001`:

$$S \to T001T$$
$$T \to 0T \mid 1T \mid \varepsilon$$

Note that here, in contrast to the previous example, we actually take advantage of the fact that the two occurrences of $T$ in the first rule are not tied together. This allows us to derive different strings on either side of `001`, which is what we want. □

The languages in the above example are all regular. In fact, we can show that all regular languages are context-free.

**Theorem 6.10** *All regular languages are context-free.*

**Proof** The proof is constructive: we give an algorithm that converts any regular expression $R$ into an equivalent CFG $G$.

$$
\begin{aligned}
1 : S_1 &\to 1 \\
0 : S_2 &\to 0 \\
0 \cup 1 : S_3 &\to S_1 \mid S_2 \\
(0 \cup 1)^* : S_4 &\to S_3 S_4 \mid \varepsilon \\
1(0 \cup 1)^* : S_5 &\to S_1 S_4
\end{aligned}
$$

Figure 6.1: Converting $1\Sigma^* = 1(0 \cup 1)^*$ into a CFG

The algorithm has six cases, based on the form of $R$. If $R = a$, then $G$ contains a single rule: $S \to a$. If $R = \varepsilon$, then $G$ again contains a single rule: $S \to \varepsilon$. If $R = \emptyset$, then $G$ simply contains no rules.

The last three cases are when $R = R_1 \cup R_2$, $R = R_1 R_2$ and $R = (R_1)^*$. These cases are recursive. First, convert $R_1$ and $R_2$ into grammars $G_1$ and $G_2$, making sure that the two grammars have no variables in common. (Rename variables if needed.) Let $S_1$ and $S_2$ be the start variables of $G_1$ and $G_2$, respectively. Then, for the case $R = R_1 \cup R_2$, $G$ contains all the variables and rules of $G_1$ and $G_2$ plus a new start variable $S$ and the following rule: $S \to S_1 \mid S_2$. The other cases are similar except that the extra rule is replaced by $S \to S_1 S_2$ and $S \to S_1 S \mid \varepsilon$, respectively. □

**Example 6.11** Consider the regular expression $1\Sigma^* = 1(0 \cup 1)^*$. Let's convert this regular expression to a CFG.

Figure 6.1 shows the necessary steps. The resulting grammar is the combination of all there rules. The start variable is $S_5$. □

**Example 6.12** Over the alphabet of parentheses, $\{ (, ) \}$, consider the language of strings that are properly nested. Given our background in computer science

and mathematics, we should all have a pretty clear intuitive understanding of what these strings are.[1] But few of us have probably thought of a precise definition, or of the need for one. But a precise definition is needed, to build compilers, for example.

A precise definition can be obtained as follows. Suppose that $w$ is a string of properly nested parentheses. Then the first symbol of $w$ must be a left parenthesis that "matches" a right parenthesis that occurs later in the string. Between these two matching parentheses, all the parentheses should be properly nested (among themselves). And to the right of the right parenthesis that matches the first left parenthesis of $w$, all the parentheses should also be properly nested. In other words, $w$ should be of the form $(u)\,v$ where $u$ and $v$ are strings of properly nested parentheses. Note that either $u$ or $v$ may be empty.

The above is the core of a recursive definition. We also need a base case. That is, we need to define the shortest possible strings of properly nested parentheses. Let's say that it's the empty string. (An alternative would be the string `()`.)

Putting it all together, we get that a string of properly nested parentheses is either $\varepsilon$ or a string of the form $(u)\,v$ where $u$ and $v$ are strings of properly nested parentheses.

A CFG that derives these strings is easy to obtain:

$$S \to (S)\,S \mid \varepsilon$$

It should be clear that this grammar is correct because it simply paraphrases the definition. (A formal proof would proceed by induction on the length of a string, for one direction, and on the length of a derivation, for the other.) □

[1] To paraphrase one of the most famous phrases in the history of the U.S. Supreme Court, "We know one when we see it."

## Exercises

6.3.1. Give CFG's for the following languages. In all cases, the alphabet is $\{0,1\}$.

a) The language $\{0^n10^n \mid n \geq 0\}$.

b) The language of strings of the form $ww^{\mathscr{R}}$.

c) The language $\{0^n1^{2n} \mid n \geq 0\}$.

d) The language $\{0^i1^j \mid i \leq j\}$.

6.3.2. Give a CFG for the language $\{1^i\#1^j\#1^{i+j}\}$. The alphabet is $\{1,\#\}$.

6.3.3. By using the algorithm of this section, convert the following regular expressions into CFG's.

a) $0^* \cup 1^*$.

b) $0^*10$.

c) $(11)^*$.

6.3.4. Consider the language of properly nested strings of parentheses, square brackets (`[`, `]`) and braces (`{`, `}`). Give a precise definition and a CFG for this language.

6.3.5. Suppose that we no longer consider that $\varepsilon$ is a string of properly nested parentheses. In other words, we now consider that the string `()` is the shortest possible string of properly nested parentheses. Give a revised definition and CFG for the language of properly nested parentheses.

## 6.4 Ambiguity and Parse Trees

It is easy to show that the language of properly nested parentheses is not regular. Therefore, the last example of the previous section shows that CFG's can express an aspect of programming languages that cannot be expressed by regular expressions. In this section, we consider another such example: arithmetic expressions. This will lead us to consider parse trees and the concept of ambiguity.

Consider the language of valid arithmetic expressions that consist of the operand `a`, the operators `+` and `*`, as well as parentheses. For example, `a+a*a` and `(a+a)*a` are valid expressions. As these examples show, expressions do not have to be fully parenthesized.

This language can be defined recursively as follows: `a` is an expression and if $e_1$ and $e_2$ are expressions, then $e_1$`+`$e_2$, $e_1$`*`$e_2$ and `(`$e_1$`)` are expressions.

As was the case for strings of properly nested parentheses, a CFG can be obtained directly from this recursive definition:

$$E \rightarrow E\texttt{+}E \mid E\texttt{*}E \mid \texttt{(}E\texttt{)} \mid \texttt{a}$$

For example, in this grammar, the string `(a+a)*a` can be derived as follows:

$$E \Rightarrow E\texttt{*}E \Rightarrow \texttt{(}E\texttt{)}\texttt{*}E \Rightarrow \texttt{(}E\texttt{+}E\texttt{)}\texttt{*}E \overset{*}{\Rightarrow} (a+a)*a$$

A derivation can be represented by a *parse tree* in which every node is either a variable or a terminal. The leaves of the tree are labeled by terminals. The non-leaf nodes are labeled by variables. If a node is labeled by variable $A$, then its children are the symbols on the right-hand-side of one of the rules for $A$.

For example, the above derivation corresponds to the parse tree shown in Figure 6.2. This parse tree is useful because it helps up visualize the derivation.

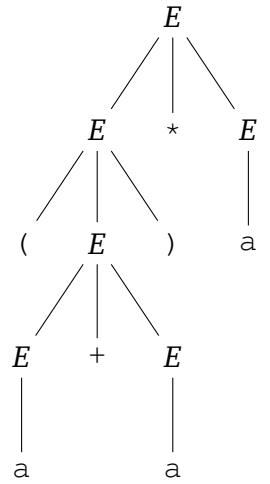



Figure 6.2: A parse tree for the string `(a+a)*a`

But in the case of arithmetic expressions, the parse tree also indicates how the expression should be evaluated: once the values of the operands are known, the expression can be evaluated by moving up the tree from the leaves. Note that interpreted in this way, the parse tree of Figure 6.2 does correspond to the correct evaluation of the expression `(a+a)*a`. In addition, this parse tree is unique: in this grammar, there is only one way in which the expression `(a+a)*a` can be derived (and evaluated).

In contrast, consider the expression `a+a*a`. This expression has two different parse trees, as shown in Figure 6.3. Each of these parse trees corresponds to a different way of deriving, and evaluating, the expression. But only the parse tree on the right corresponds to the correct way of evaluating this expression because it follows the rule that multiplication has precedence over addition.

In general, a parse tree is viewed as assigning *meaning* to a string. When

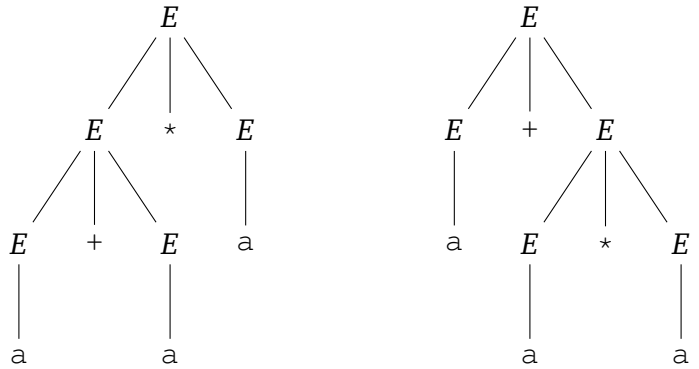



Figure 6.3: Parse trees for the string a+a*a

a string has more than one parse tree in a particular grammar, then the grammar is said to be *ambiguous*. The above grammar for arithmetic expressions is ambiguous.

In practical applications, we typically want unambiguous grammars. In the case of arithmetic expressions, an unambiguous CFG can be designed by creating *levels* of expressions: expressions, terms, factors, operands. Here's one way of doing this:

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid \texttt{a}$$

In this grammar, the string a+a*a has only one parse tree, the one shown in Figure 6.4. And this parse tree does correspond to the correct evaluation of the expression.

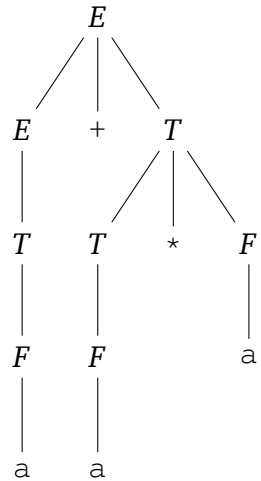



Figure 6.4: Parse tree for the string `a+a*a`

It is not hard to see that this last grammar is unambiguous and that it correctly enforces precedence rules as well as left associativity. Note, however, that some context-free languages are *inherently ambiguous*, in the sense that they can only be generated by ambiguous CFG's.

## Study Questions

6.4.1. What is a parse tree?

6.4.2. What does it mean for a grammar to be ambiguous?

6.4.3. What does it mean for a CFL to be inherently ambiguous?

### Exercises

6.4.4. Consider the expression a⋆a+a. Give two different parse trees for this string in the first grammar of this section. Then give the unique parse tree for this string in the second grammar of this section.

## 6.5 A Pumping Lemma

We know that $\{0^n1^n \mid n \geq 0\}$ is context-free because it is generated by the grammar

$$S \to 0S1 \mid \varepsilon$$

This grammar generates the 0's and 1's in pairs, starting from the outside and progressing towards the middle of the string:

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111$$

Now consider the language $\{a^nb^nc^n \mid n \geq 0\}$. One idea would be a rule such as $S \to aSbSc$ but that would mix the different kinds of letters. Another idea would be $S \to T_1T_2$ where $T_1$ derives strings of the form $a^{2n}b^n$ while $T_2$ derives strings of the form $b^nc^{2n}$. But that would not ensure equal numbers of a's and c's.

It turns out that $\{a^nb^nc^n\}$ is not context-free: there is no CFG that can generate it. And, as in the case of regular languages, this can be shown by using a Pumping Lemma.

**Theorem 6.13 (Pumping Lemma)** *If $L$ is a CFL, then there is a number $p > 0$, called the* pumping length*, such that if $w \in L$ and $|w| \geq p$, then $w$ can be written*

*as $uvxyz$ where*

1. $|vxy| \leq p$,
2. $vy \neq \varepsilon$,
3. $uv^k xy^k z \in L$, *for every* $k \geq 0$.

**Example 6.14** Suppose that $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is context-free. Let $p$ be the pumping length. Consider the string $w = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, $w$ can be written as $uvxyz$ where

1. $|vxy| \leq p$.
2. $vy \neq \varepsilon$.
3. $uv^k xy^k z \in L$, for every $k \geq 0$.

There are two cases to consider. First, suppose that either $v$ or $y$ contains more than one type of symbol. Then $uv^2xy^2z \notin L$ because that string is not even in $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$.

Second, suppose that $v$ and $y$ each contain only one type of symbol. Then $uv^2xy^2z$ contains additional occurrences of at least one type symbol but not of all three types. Therefore, $uv^2xy^2z \notin L$.

In both cases, we have that $uv^2xy^2z \notin L$. This is a contradiction and proves that $L$ is not context-free. □

Note that the condition $|vxy| \leq p$ was not used in this proof that $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\}$ is not context-free. Our next example will require that we use that condition.

Let $L$ be the language of strings of the form $ww$. An exercise in the previous chapter asked you to show that the language of strings of the form $ww^{\mathscr{R}}$

is context-free. The idea is to generate the symbols of $ww^{\mathcal{R}}$ starting from the outside and progressing towards the middle:

$$S \rightarrow 0S0 \mid 1S1 \mid \varepsilon$$

But this idea does not work with strings of the form $ww$ because the first symbol of the string must match a symbol located in its middle.

Instead, we now use the Pumping Lemma to show that the language of strings of the form $ww$ is not context-free.

**Example 6.15** Let $L$ be the language of strings of the form $ww$. Suppose that $L$ is context-free. Let $p$ be the pumping length. Consider the string $w = 0^p1^p0^p1^p$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, $w$ can be written as $uvxyz$ where

1. $|vxy| \leq p$.
2. $vy \neq \varepsilon$.
3. $uv^kxy^kz \in L$, for every $k \geq 0$.

The string $w$ consists of four blocks of length $p$. Since $|vxy| \leq p$, $v$ and $y$ can touch at most two blocks. Suppose that $v$ and $y$ are both contained within a single block, the first one, for example. Then $uv^2xy^2z = 0^{p+i}1^p0^p1^p$, where $i > 1$. This string is clearly not in $L$. The same is true for the other blocks.

Now suppose that $v$ and $y$ touch two consecutive blocks, the first two, for example. Then $uv^0xy^0z = 0^i1^j0^p1^p$ where $0 < i, j < p$. Once again, this string is clearly not in $L$. The same is true for the other blocks.

Therefore, in all cases, we have that $w$ cannot be pumped. This contradicts the Pumping Lemma and proves that $L$ is not context-free. □

### Exercises

6.5.1. Use the Pumping Lemma to show that the language $\{0^n1^n0^n \mid n \geq 0\}$ is not context-free.

6.5.2. Use the Pumping Lemma to show that the language $\{a^ib^jc^k \mid i \leq j \leq k\}$ is not context-free. The alphabet is $\{a, b, c\}$.

6.5.3. Use the Pumping Lemma to show that the language $\{1^i\#1^j\#1^{i+j} \mid i \leq j\}$ is not context-free. The alphabet is $\{1, \#\}$.

6.5.4. Consider the language of strings of the form $ww^{\mathcal{R}}$ that also contain equal numbers of 0's and 1's. For example, the string 0110 is in this language but 011110 is not. Use the Pumping Lemma to show that this language is not context-free.

6.5.5. Use the Pumping Lemma to show that the language $\{1^i\#1^j\#1^{ij}\}$ is not context-free. The alphabet is $\{1, \#\}$. (Recall that Exercises 5.2.2 and 6.3.2 asked you to show that the similar language for unary addition is context-free but not regular. Therefore, when the problems are presented in this way, unary addition is context-free but not regular, while unary multiplication is not even context-free.)

## 6.6 Proof of the Pumping Lemma

We start by coming up with a plan for the proof, without worrying about justifying every detail. As long as the various steps make sense intuitively. We will then go back over this plan, make everything precise and justify every step.

In what follows, we will highlight the key ideas of the proof. Some are fairly obvious while others are somewhat clever.

Suppose that $L$ is context-free and suppose that $w$ is a long string in $L$. The Pumping Lemma says that $w$ can be written as $uvxyz$ such that

1. $vxy$ is not too long,

2. $v$ and $y$ are not both empty,

3. $uv^kxy^kz \in L$, for every $k \geq 0$.

Our main goal is to be able to pump $w$. In the case of regular languages, this was achieved by looking for a repetition in the sequence of states that a DFA goes through while reading $w$.

It makes sense to try something similar here. So the first idea of our proof is: *look for a repetition*. But a repetition of what and where?

Let $G$ be a CFG for $L$. Since $w$ is long, it makes sense to think that $G$ requires many steps to derive $w$. And since each step involves at least one variable and since there are only so many variables in $G$, there must be two different steps that involve the same variable. So every derivation of $w$ must look something like this:

$$S \stackrel{*}{\Rightarrow} u_1Au_2 \stackrel{*}{\Rightarrow} v_1Av_2 \stackrel{*}{\Rightarrow} w.$$

In this derivation, $A$ is the repeated variable.

Now, there are two ways in which this repetition can occur: it could be that the second $A$ is derived from the first $A$, or that the second $A$ is derived from a variable in either $u_1$ or $u_2$, the strings to either side of the first $A$.

This is best illustrated by looking at parse trees. Figure 6.5 illustrates the first type of repetition, the one where the second $A$ is derived from the first $A$. We call this a *nested* repetition. In the first tree of Figure 6.5, $u$ is the string that is eventually derived from $u_1$, $y_1$ is the string derived from the first $A$ to the left of the second $A$, and $v$ is the string eventually derived from $y_1$. Similarly for $y_2$, $y$ and $z$. The string $x$ is the one eventually derived from the second $A$. The

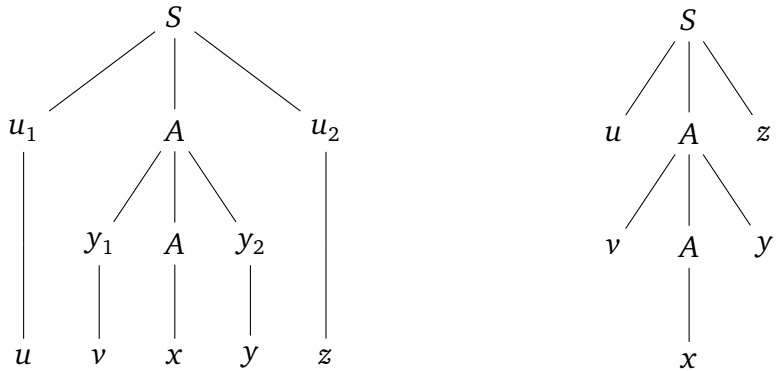



Figure 6.5: A nested repetition

second tree in that figure represents the same derivation but without showing the intermediate strings. (Note that, strictly speaking, these trees are not parse trees but high-level outlines of parse trees.)

Figure 6.6 illustrates the second type of repetition, the *non-nested* type, where the second $A$ is derived from a variable in either $u_1$ or $u_2$. The case that's illustrated is the one where the second $A$ is derived from a variable in $u_2$. Once again, the second tree represents the same derivation but without showing the intermediate strings.

Recall that we are looking for a way to pump $w$. It's not clear how a non-nested repetition would allow us to pump $w$. But a nested repetition does. That's because the portion of the parse tree that derives $vAy$ from the first $A$ in Figure 6.5 can be omitted or repeated as illustrated in Figure 6.7. In this figure, the cases $k = 0$ and $k = 2$ are shown but it should be clear that for every $k \geq 0$, the string $uv^k xy^k z$ can be derived in a similar way.

Now, how can we ensure that a derivation of $w$ contains a nested repetition?

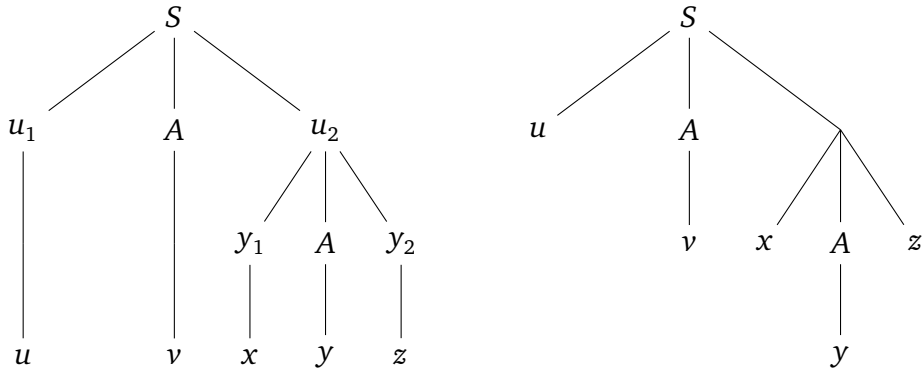



Figure 6.6: A non-nested repetition

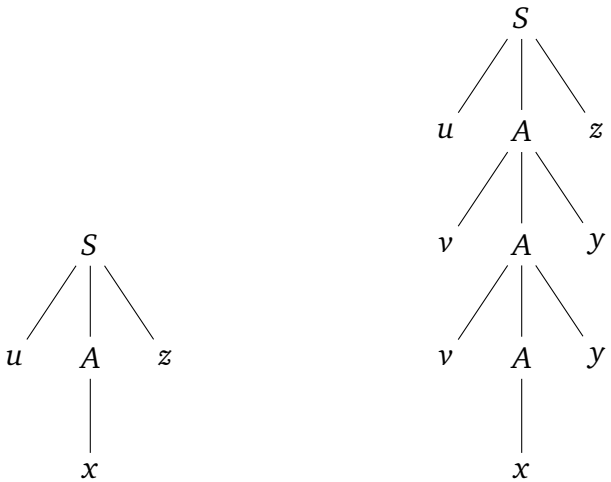



Figure 6.7: Pumping with a nested repetition

The key observation here is to notice that in a nested repetition such as the one illustrated in Figure 6.5, the two $A$'s occur along the same path down the tree. So this is the second key idea of the proof: don't look for any repetition in a derivation, *look for a repetition along a single path of a parse tree*.

How do we know such a repetition even exists? Since $w$ is long, it makes sense that any parse tree for $w$ would not only have to be large, but would also have to be deep, that is, that it would need to contain a long path. And once again, since the number of variables in $G$ is limited, a long enough path must contain a repeated variable.

So we think we know how to show that a long $w$ can be pumped. What about the other two conditions of the Pumping Lemma?

One of them states that $v$ and $y$ are not both empty. How can ensure that this is the case. Well, suppose it's not. That is, suppose that $v$ and $y$ are both empty. This would imply that the string derived in the first tree of Figure 6.7, the one that corresponds to pumping down, is identical to $w$. But then, this tree would be a parse tree for $w$ that's smaller than the original one, which is shown again in Figure 6.8. We can avoid this situation by simply *choosing to focus, from the start, on the smallest possible parse tree for* $w$. This is the third key idea of our proof. In a smallest parse tree for $w$, $v$ and $y$ cannot both be empty since otherwise we would be able to construct a smaller parse tree for $w$.

The other condition is that $vxy$ is not too long. Recall that, as illustrated in Figure 6.8, $vxy$ is the string ultimately derived from the first $A$. We said earlier that if a string is long, then its parse should have to be deep. Therefore, it should also be true that if a tree is shallow, then the string it derives is short. So a way to ensure that $vxy$ is not too long is to ensure that the tree rooted at the first $A$ is not too deep.

How can this be achieved? The answer is, by *looking for a repetition while going up a long path in the parse tree for* $w$. This is the fourth key idea of the

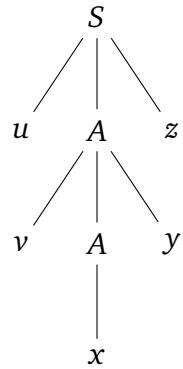



Figure 6.8: The original parse tree for $w$

proof. Since the number of variables in $G$ is limited, a repetition will be found without having to go too high up that path. And this ensures that the tree rooted at the first $A$ is not too deep and that $vxy$ is not too long.

There is a possible problem in what we just said. We know the first $A$ is not too high up that long path. But the tree rooted at that $A$ could be deeper than that if it contains another path that's longer than the first one, as illustrated in Figure 6.9. How can we make sure that this doesn't happen?

The answer is the fifth and final key idea of the proof: by *choosing to focus, from the start, on one of the longest possible paths in the parse tree*. As explained above, we then go up that path until we find a repeated variable $A$. The first $A$ is not too high up that path. If the tree rooted at the first $A$ contains a path that's longer than that, as illustrated in Figure 6.9, then that second path, together with the path that leads from the top of the tree to the first $A$ would be longer than the first path. This cannot happen. Therefore, every path in the tree rooted at the first $A$ is not too long, which means that this tree is not too deep and implies

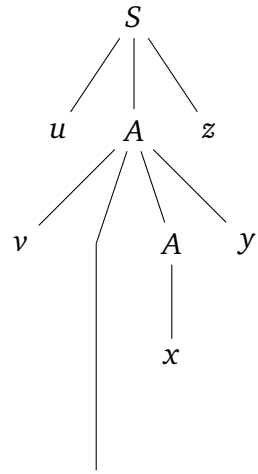



Figure 6.9: A longer path in the tree rooted at the first $A$

that $vxy$ is not too long.

We now seem to have all the key ideas needed for a proof of the Pumping Lemma. What we need to do next is make this all more precise and justify every step.

In particular, we need to determine how long $w$ needs to be to guarantee that a parse tree for $w$ contains a path with a repeated variable. Any path of length $k$ in a parse tree contains $k$ variables and a terminal (at the bottom). To ensure that a variable repeats, we need a path of length at least $|V| + 1$, where $|V|$ is the number of variables in the grammar.

The height of a tree is the length of the longest path it contains. Suppose that a parse tree doesn't contain a path of length at least $|V| + 1$. That is, suppose that a parse tree has height at most $|V|$. How long of a string can it derive? Let $b$ be the maximum number of symbols on the right-hand side of any rule in the

grammar. It's not hard to see that a parse tree of height at most $|V|$ derives a string of length at most $b^{|V|}$. Therefore, if $|w| \geq b^{|V|} + 1$, then any parse tree for $w$ must be of height at least $|V| + 1$, which implies that it contains at least one path of length at least $|V| + 1$.

Later in the proof, we consider one of the longest paths in such a tree and find a repeated variable $A$ by going up that path. How far do we need to go? At most $|V| + 1$ steps. As explained earlier, this implies that the tree rooted at the first $A$ cannot contain a path longer than that. In other words, that tree has height at most $|V| + 1$ and the string derived by that tree, which is $xvy$, is no longer than $b^{|V|+1}$.

So for the proof to work, we need the length of $|w| \geq b^{|V|} + 1$ and this results in $|xvy| \leq b^{|V|+1}$. If we choose $p = \max(b^{|V|} + 1, b^{|V|+1})$, then $|w| \geq p$ would work and we would get $|xvy| \leq p$.

We are finally ready for a clean write-up of the entire proof.

**Proof** Let $L$ be a CFL and let $G$ be a CFG that generates $L$. Let $b$ be the maximum number of symbols on the right-hand side of any rule of $G$. Let $p = \max(b^{|V|} + 1, b^{|V|+1})$.[2] Now suppose that $w$ is any string in $L$ with $|w| \geq p$.

Consider one of the smallest possible parse trees for $w$. Because $|w| \geq b^{|V|}+1$, the height of that tree must at least $|V|+1$. Let $\pi$ to be one of the longest possible paths in the tree. Then the length of $\pi$ is at least $|V| + 1$, which implies that $\pi$ contains a repetition among its bottom $|V| + 1$ variables. Let $A$ be one such repeated variable. Then the tree is of the form shown in Figure 6.8.

Define $u$, $v$, $x$, $y$ and $z$ as suggested by Figure 6.8. Then $w = uvxyz$ and by omitting or repeating the portion of the tree that derives $vAy$ from the first $A$, a parse tree can be constructed for every string of the form $uv^kxy^kz \in L$ with $k \geq 0$. This is illustrated in Figure 6.7.

[2]Note that if $b \geq 2$, then $p = b^{|V|+1}$.

We also have that $vy \neq \varepsilon$, since otherwise the first parse tree of Figure 6.7 would be a parse tree for $w$ that's smaller than the original tree, and that's not possible since we chose that tree to be one of the smallest possible parse trees for $w$.

In Figure 6.8, the height of the subtree rooted at the higher $A$ is at most $|V|+1$. Because if that subtree contained a path longer than $|V|+1$, as illustrated in Figure 6.9, then that path, together with the path that goes from the top of the entire tree to the higher $A$ would be longer than $\pi$, and that's not possible since we chose $\pi$ to be one of the longest possible paths in the entire tree. This implies that $|vxy| \leq b^{|V|+1} \leq p$. □

## 6.7 Closure Properties

The constructions in the proof of Theorem 6.10 imply the following:

**Theorem 6.16** *The class of context-free languages is closed under union, concatenation and the star operation.*

The details of a proof are left as an exercise.

On the other hand, while the class of regular languages was closed under both union and intersection, this is not the case with context-free languages.

**Theorem 6.17** *The class of context-free languages is* not *closed under intersection.*

**Proof** Consider the language $L_1 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^i \mid i, n \geq 0\}$. This language is context-free:

$$
\begin{aligned}
S &\to TU \\
T &\to \mathtt{a}T\mathtt{b} \mid \varepsilon \\
U &\to \mathtt{c}U \mid \varepsilon
\end{aligned}
$$

Now, consider the language $L_2 = \{\mathtt{a}^i\mathtt{b}^n\mathtt{c}^n \mid i, n \geq 0\}$. This language too is context-free:

$$S \to UT$$
$$T \to \mathtt{b}T\mathtt{c} \mid \varepsilon$$
$$U \to \mathtt{a}U \mid \varepsilon$$

But $L_1 \cap L_2 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ and we now know that this language is not context-free. Therefore, the class of CFL's is not closed under intersection. □

**Corollary 6.18** *The class of context-free languages is not closed under complementation.*

**Proof** If it was, then it would be closed under intersection because of De Morgan's Law: $A \cap B = \overline{\overline{A} \cup \overline{B}}$. □

What about a concrete example of a CFL whose complement is not context-free? Here's one. We know that $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is not context-free. The complement of this language is

$$\overline{\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*} \cup \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i \neq j \text{ or } j \neq k\}.$$

It's not hard to show that this language is context-free (see the exercises).

Here's another example. We know that the language of strings of the form $ww$ is not context-free. The complement of this language is

$$\{w \in \{\mathtt{0},\mathtt{1}\}^* \mid |w| \text{ is odd}\} \cup \{xy \mid x, y \in \{\mathtt{0},\mathtt{1}\}^*, |x| = |y| \text{ but } x \neq y\}$$

This language is also context-free (see the exercises).

### Exercises

6.7.1. Prove Theorem 6.16.

6.7.2. Give a CFG for the language $\{a^i b^j c^k \mid i \neq j \text{ or } j \neq k\}$. Show that this implies that the complement of $\{a^n b^n c^n \mid n \geq 0\}$ is context-free.

6.7.3. Give a CFG for the language $\{xy \mid x, y \in \{0,1\}^*, |x| = |y| \text{ but } x \neq y\}$. Show that this implies that the complement of the language of strings of the form $ww$ is context-free.

6.7.4. Give a CFG for the language $\{x\#y \mid x, y \in \{0,1\}^* \text{ and } x \neq y\}$. Use this to construct another example of a CFL whose complement is not context-free.

## 6.8 Pushdown Automata

The class of CFL's is not closed under intersection but it is closed under intersection with a regular language:

**Theorem 6.19** *If $L$ is context-free and $B$ is regular, then $L \cap B$ is context-free.*

It's not clear how this could be proved using CFG's: how can a CFG be combined with a regular expression to produce another CFG?

Instead, recall that for regular languages, closure under intersection was proved by using DFA's and the pair construction. To carry out a similar proof for the intersection of a CFL and a regular language, we would need to identify a class of automata that corresponds exactly to CFL's.

The key is to use nondeterminism. Here's a first idea. Suppose that $G$ is a CFG with start variable $S$. We want an algorithm that given a string $w$ as input,

```
set u = S
while (u contains at least one variable)
    let A be any variable in u
    nondeterministically choose a rule for A
    replace A by the right-hand side of that rule

if (u == w)
    accept
else
    reject
```

Figure 6.10: An algorithm that simulates a CFG

determines if $w$ can be derived in $G$. The idea is to have the algorithm construct a derivation for $w$. At any moment, the algorithm stores the string $u$ of variables and terminals it has derived from $S$ so far. Initially, $u = S$. At each step, the algorithm replaces one of the variables in $u$ by the right-hand-side of a rule. The rule is chosen nondeterministically. When all the variables in $u$ have been replaced by terminals, the algorithm verifies that $u = w$. If yes, the algorithm found a way to derive $w$ in $G$. If not, the derivation that the algorithm tried did not work. Figure 6.11 shows a nondeterministic algorithm that simulates $G$.

This nondeterministic algorithm works because if $w$ can be derived in $G$, then there is a derivation that works and the algorithm will find it. On the other hand, if $w$ cannot be derived in $G$, then there is no derivation that works and the algorithm will reject no matter what it tries.

The algorithm of Figure 6.10 can be described as a *single-scan, nondeterministic algorithm*. We will learn how to precisely define this class of algorithms later in these notes but they will turn out to be too powerful for CFG's: the class of languages these algorithms recognize contains not only all CFL's but also lan-

guages that are not context-free. So we need to somehow restrict the algorithm of Figure 6.10 to identify a type of algorithm that recognizes fewer languages.

The idea is to restrict the type of memory the algorithm uses. Modify the algorithm so it always works on the leftmost symbol of $u$. If the leftmost symbol of $u$ is a variable, apply a rule to it. If the leftmost symbol of $u$ is a terminal, that terminal must be the first terminal of the input. So the algorithm will verify that this is the case and then remove that symbol from $u$ and move to the next symbol in the input. The result is that in the algorithm of Figure 6.10, $u$ was the string of variables and terminals that had been derived so far from $S$. In the new algorithm, the string of variables and terminals that has been derived so far from $S$ is $xu$, where $x$ is the portion of $w$ that has already been read.

The advantage of this modification is that, as mentioned before, the algorithm only needs to access the leftmost symbol of $u$. In other words, $u$ can be stored in a *stack*, with the left end of $u$ at the top of the stack. Figure 6.11 shows the resulting algorithm.

Here's an example of how this algorithm works. Consider the following grammar:

$$E \to E{+}T \mid T$$
$$T \to T \star F \mid F$$
$$F \to (E) \mid \mathtt{a}$$

Figure 6.12 shows a *trace* of this algorithm as it runs on input `a+a*a`.

The new algorithm works because if $w$ can be derived in $G$, then there is a derivation that works, the algorithm will find it and, eventually, the stack will be empty and all of the input will have been read, which will cause the algorithm to accept. And the reverse is also true: if the algorithm accepts, it's because the stack is empty and all of the input has been read, which can only happen if the

```
push S on the stack
while (the stack is not empty)
    if (the top of the stack is a variable)
        nondeterministically choose a rule for that
            variable
        replace the variable by the right-hand side of
            that rule (with the left end at the top)
    else  // the top of the stack is a terminal
        if (the end of the input has been reached)
            reject
        read next input symbol c
        if (the top of the stack equals c)
            pop the stack
        else
            reject

if (the end of the input has been reached)
    accept
else
    reject
```

Figure 6.11: A stack algorithm that simulates a CFG

| input (unread portion) | stack |
|---|---|
| a+a*a | E |
| a+a*a | E+T |
| a+a*a | T+T |
| a+a*a | F+T |
| a+a*a | a+T |
| +a*a | +T |
| a*a | T |
| a*a | T*F |
| a*a | F*F |
| a*a | a*F |
| *a | *F |
| a | F |
| a | a |
| <empty> | <empty> |

Figure 6.12: A sample run of the algorithm that simulates CFG's

algorithm found a derivation that works.

The algorithm of Figure 6.11 can be described as a *single-scan, nondeterministic stack algorithm*. We won't do it in these notes but these algorithms can be formalized as a type of automaton called a *pushdown automaton* (PDA). PDA's are essentially NFA's augmented with a stack.

It can be shown that PDA's are equivalent to CFG's: every CFG can be simulated by a PDA, and every PDA has an equivalent CFG. Therefore, PDA's characterize CFL's just like DFA's and NFA's characterize regular languages.

**Theorem 6.20** *A language is context-free if and only if it is recognized by some PDA.*

In addition, as was the case with regular languages, the proof of this theorem is constructive: there are algorithms for converting CFG's into PDA's, and vice versa.

We can now sketch the proof that the class of CFL's is closed under intersection with a regular language.

**Proof** (*Of Theorem 6.19. Sketch.*) Suppose that $L$ is context-free and $B$ is regular. By Theorem 6.20, let $P$ be a PDA for $L$. Let $M$ be a DFA for $B$. The pair construction can be used to combine the "NFA" of $P$ with $M$. This results in a new PDA that recognizes $L \cap B$. By Theorem 6.20, this implies that $L \cap B$ is context-free. □

Note that the above proof sketch does not work for the intersection of two CFL's. The problem is that combining two PDA's would produce an automaton with two stacks and this is not a PDA. In fact, adding a second stack to PDA's produces a class of automata that can recognize languages that are not context-free. Here's an example.

**Example 6.21** Consider the language $\{a^n b^n c^n \mid n \geq 0\}$. A two-stack, single-scan algorithm that recognizes this language can be designed as follows. The idea is to push all the a's onto one stack, all the b's onto the other stack, and then pop both stacks once for every c in the input. We then accept if and only if both stacks become empty at the same time as the end of the input is reached. This algorithm can be described in pseudocode as shown in Figure 6.13. □

We end this section with another example. If a language is context-free, as we learned in this section, a PDA for this language can be obtained by converting a CFG into a PDA. But sometimes it is just as easy to design a PDA directly. The following example illustrates this idea. It is also an interesting demonstration of the power of nondeterminism.

**Example 6.22** Consider the language $ww^{\mathcal{R}}$. (Recall that $w^{\mathcal{R}}$ is the reverse of $w$.) A single-scan, nondeterministic stack algorithm that recognizes this language can be designed as follows. The idea is to push all the symbols from the first half of the input onto the stack. Then, as the symbols from the second half of the input are read, they are matched, one by one, with the symbols contained in the stack. For example, if the input is 0110, then 01 would be pushed on the stack, with 0 at the bottom and 1 at the top. Then, the second half of the string, 10, would be read and match the symbols from the stack.

One difficulty with this idea is determining where the middle of the string is. This is where nondeterminism is useful. The algorithm can simply guess where the middle is. If the guess is correct, the end of the input will be reached at exactly the same time as the stack becomes empty. Otherwise, the guess is incorrect and the algorithm would reject. This algorithm works because if the input is in the language, there is a correct guess that will cause the algorithm to accept. But if the input is not in the language, then the algorithm will always reject because even if it guesses the middle correctly, the algorithm will find a

```
if (end of input) // input is empty
    accept
initialize both stacks to empty
read next char x
while (x is a)
    push a on the first stack
    if (end of input) // some a's but no b's or c's
        reject
    read next char x
while (x is b)
    push b on the second stack
    if (end of input) // some b's but no c's
        reject
    read next char x
while (x is c)
    if (either stack is empty)
        // more c's than either a's or b's
        reject
    pop both stacks
    if (end of input)
        if (both stacks empty)
            accept
        else
            reject  // more a's or b's than c's
    read next char x
reject  // a's after b's, or either a's or b's after c's
```

Figure 6.13: A two-stack algorithm for the language $\{a^n b^n c^n \mid n \geq 0\}$

```
initialize stack to empty

while (true)
    read next char x
    push x on the stack
    nondeterministically chose to either continue with
        the loop or exit the loop

while (not end of input)
    read next char x
    if (stack is empty) // middle guessed too early
        reject
    pop y from the stack
    if (x != y)
        reject

if (stack empty)
    accept
else
    reject  // middle guessed too late
```

Figure 6.14: A stack algorithm for the language $ww^{\mathcal{R}}$

mismatch and reject. The entire algorithm can be described in pseudocode as shown in Figure 6.14. □

## Study Questions

6.8.1. What is a PDA?

### Exercises

6.8.2. Describe a single-scan stack algorithm that recognizes the language $\{0^n1^n \mid n \geq 0\}$. (Note: nondeterminism is not needed for this language.)

6.8.3. Describe a two-stack, single-scan, nondeterministic algorithm that recognizes the language of strings of the form $ww$.

## 6.9 Deterministic Algorithms for CFL's

In the previous section, we saw that every CFL can be recognized by a single-scan, nondeterministic stack algorithm. We also said that these algorithms can be formalized as PDA's. In this section, we will briefly discuss the design of *deterministic* algorithms for CFL's. Our discussion will be informal.

To be more precise, we are interested in showing that for every CFG $G$, there is an algorithm that given a string of terminals $w$, determines if $G$ derives $w$. A derivation is simply a sequence of rules. Since $G$ only has a finite number of rules, derivations can be generated just like strings, starting with the shortest possible derivation. So a simple idea for an algorithm is to generate every possible sequence of rules and determine if any of them constitutes a valid derivation of $w$.

If $w \in L(G)$, then this algorithm will eventually find a derivation of $w$. But if $w \notin L(G)$, then no derivation will ever be found and the algorithm may go on forever.

A way to fix this problem would be to know when to stop searching: to know a length $r$, maybe dependent on the length of $w$, with the property that if $w$ can be derived, then it has at least one derivation of length no greater than $r$. A difficulty in finding such a number $r$ is that in some grammars, the shortest derivation of a string can be needlessly long. For example, in a grammar

that contains $\varepsilon$ rules, the shortest way to derive a particular string may be to introduce large numbers of variables that are later deleted. Unit rules (rules that have a single variable on the right-hand side) are similarly problematic.

Fortunately, it turns out that it is always possible to eliminate $\varepsilon$ rules and unit rules from a CFG. In fact, every CFG can always be transformed into an equivalent CFG that's in *Chomsky Normal Form*:

**Definition 6.23** *A CFG $G = (V, \Sigma, R, S)$ is in* Chomsky Normal Form (CNF) *if every one of its rules is in one of the following three forms:*

$$A \to BC, \text{where } A, B, C \in V \text{ and } B, C \neq S$$
$$A \to a, \text{where } A \in V \text{ and } a \in \Sigma$$
$$S \to \varepsilon$$

**Theorem 6.24** *Every CFG has an equivalent CFG in CNF.*

We won't prove this theorem in these notes but note that the theorem can be proved constructively: there is an algorithm that given a CFG $G$, produces an equivalent CFG $G'$ in CNF.

CFG's in CNF have the following useful property:

**Theorem 6.25** *If $G$ is a CFG in CNF and if $w$ is a string of terminals of length $n > 0$, then every derivation of $w$ in $G$ as length $2n - 1$.*

**Proof** Suppose that $D$ is a derivation of $w$. Since $w$ is of length $n$, $D$ must contain exactly $n$ instances of rules of the second form. Each rule of the first form introduces one new variable to the derivation. Since $S$ can't appear on the right-hand-side of those rules, $D$ must contain exactly $n-1$ rules of the first form. □

Therefore, the number we were looking for is $2n-1$, where $n = |w|$. This gives the following algorithm for any CFG $G$ in CNF:

1. Let $w$ be the input and let $n = |w|$.

2. Generate every possible derivation of length $2n-1$ and determine if any of them derives $w$.

Since every CFG has an equivalent CFG in CNF, this shows that every CFG has an algorithm.

The above algorithm is correct but very inefficient. For example, consider any CFL in which the number of strings of length $n$ is $2^{\Omega(n)}$. Each of those strings has a different derivation. Therefore, given a string of length $n$, in the worst case, the algorithm will need to examine $2^{\Omega(n)}$ derivations.

A much faster algorithm can be designed by using the powerful technique of *dynamic programming*.[3] This algorithm runs in time $\Theta(n^3)$.

For many practical applications, such as the design of compilers, this is still too slow. In these applications, it is typical to focus on *deterministic context-free languages* (DCFL's). DCFL's are the languages that are recognized by *deterministic* PDA's (DPDA's). In other words, DCFL's are CFL's for which nondeterminism is not needed.

Note that every regular language is a DCFL because DPDA's can simulate DFA's. But not all CFL's are DCFL's. And it's not hard to see why. The class of DCFL's is closed under complementation. This is not surprising since deterministic automata normally allow us to "switch" the accepting and non-accepting status of their states. We did that with DFA's to show that the class of regular languages is closed under complementation. In the last section, we showed

[3] At Clarkson, this technique is covered in the courses CS344 *Algorithms and Data Structures* or CS447/547 *Computer Algorithms*.

that the class of CFL's is not closed under complementation. This immediately implies that these the class of DCFL's cannot be equal to the class of CFL's.

We can even come up with concrete examples of CFL's that are not DCFL's. The language $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is not context-free. But, in the previous section, we saw that the complement of $L$ is context-free. If $\overline{L}$ was a DCFL, then $L$ would also be a DCFL, contradicting the fact that $L$ is not even context-free. Therefore, $\overline{L}$ is a CFL but not a DCFL.

Note that there are restrictions of CFG's that are equivalent to DPDA's and generate precisely the class of DCFL's. This topic is normally covered in detail in a course that focuses on the design of compilers.[4]

## Study Questions

6.9.1. What forms of rules can appear in a grammar in CNF?

## Exercises

6.9.2. A *leftmost derivation* is one where every step replaces the leftmost variable of the current string of variables and terminals. Consider the following modification of the algorithm described in this section

1. Let $w$ be the input and let $n = |w|$.
2. Generate every possible leftmost derivation of length $2n-1$ and determine if any of them derives $w$.

Show that this algorithm runs in time $2^{O(n)}$. *Hint*: The similar algorithm described earlier in this section does not necessarily run in time $2^{O(n)}$.

[4] At Clarkson, this is CS445/545 *Compiler Construction*.

6.9.3. Consider the language of strings of the form $ww$. Show that the complement of this language is not a DCFL.

# Chapter 7

# Turing Machines

## 7.1 Introduction

Recall that one of our goals in these notes is to show that certain computational problems cannot be solved by any algorithm whatsoever. As explained in Chapter 1 of these notes, this requires that we define precisely what we mean by an algorithm. In other words, we need a *model of computation*. We'd like that model to be simple so we can prove theorems about it. But we also want our model to be relevant to real-life computation so that our theorems say something that applies to real-life algorithms.

In Section 2.1, we described the standard model: the Turing machine. Figure 7.1 shows a Turing machine. The *control unit* of a Turing machine consists of a transition table and a state. In other words, the control unit of a Turing machine is essentially a DFA, which means that a *Turing machine* is essentially a DFA augmented by memory.

The *memory* of a Turing machine is a string of symbols. That string is semi-infinite: it has a beginning but no end. At any moment in time, the control unit

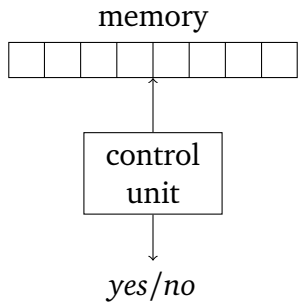



Figure 7.1: A Turing machine

has access to one symbol from the memory. We imagine that this is done with a *memory head*.

Here's an overview of how a Turing machine operates. Initially, the memory contains the input string followed by an infinite number of a special symbol called a *blank symbol*. The control unit is in its start state. Then, based on the memory symbol being scanned and the internal state of the control unit, the Turing machine overwrites the memory symbol, moves the memory head by one position to the left or right, and changes its state. This is done according to the transition function of the control unit. The computation halts when the Turing machine enters one of two special *halting states*. One of these is an accepting state while the other is a rejecting state.

It's important to note that the amount of memory a Turing machine can use is not limited by the length of the input string. In fact, there is no limit to how much memory a Turing machine can use. A Turing machine is always free to move beyond its input string and use as much memory as it wants.

In the next section, we will formally define what a Turing machine is. In the meantime, here's an informal example that illustrates what Turing machines

can do.

Consider the language $\{a^n b^n c^n \mid n \geq 0\}$. A Turing machine that recognizes this language can be designed as follows. Suppose that the input is

aaabbbccc

The basic idea is to make sure that every a has a matching b and a matching c. The Turing machine can keep track of which symbols have been matched by crossing them off. So the machine starts by crossing off the first a and then moves its memory head to the right until it finds, and crosses off, a b and a c:

xaaxbbxcc

If ever a matching b or c cannot be found, the machine rejects. The machine then keeps going this way until all the a's have been crossed off. When that happens, all it needs to do is verify that all the b's and c 's have been crossed off as well.

Here's a more complete and more detailed description of this Turing machine:

1. If the input is empty, accept.
2. Scan the input from left to right to verify that it is of the form $a^*b^*c^*$. In other words, verify that there are no a's after the first b, and that there are no a's or b's after the first c. If that's not the case, reject.
3. Return the head to the beginning of the memory.
4. Move right to the first a and cross it off.
5. Move right to the first b and cross it off. If no b is found, reject.

6. Move right to the first $\mathtt{c}$ and cross it off. If no $\mathtt{c}$ is found, reject.

7. Repeat Steps 3 to 6 until all the $\mathtt{a}$'s have been crossed off. When that happens, scan the input to verify that all other symbols have been crossed off. If so, accept. Otherwise, reject.

The above description is still missing several details. We can call it a sketch or outline of a Turing machine. We will be able to give a more precise description once we have formally defined what a Turing machine is in the next section.

Recall that the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is not context-free. So the example of this section points to the fact that Turing machines can recognize languages that both DFA's and PDA's can't.

### Exercises

7.1.1. In the style of this section, sketch the operation of a Turing machine for the language $\{\mathtt{1}^i\#\mathtt{1}^j\#\mathtt{1}^{i+j}\}$.

7.1.2. In the style of this section, sketch the operation of a Turing machine for the language $\{\mathtt{1}^i\#\mathtt{1}^j\#\mathtt{1}^{ij}\}$.

## 7.2 Formal Definition

**Definition 7.1** *A* Turing machine *is a* 7*-tuple* $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ *where*

1. $Q$ *is a finite set of* states.

2. $\Sigma$ *is an alphabet called the* input alphabet.

3. $\Gamma$ *is an alphabet called the* memory alphabet *(or* tape alphabet*)*[1] *that satisfies the following conditions:* $\Gamma$ *contains* $\Sigma$ *as well as a special symbol* $\textvisiblespace$*, called the* blank*, that is not contained in* $\Sigma$*.*

4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *is the* transition function*.*

5. $q_0 \in Q$ *is the* starting state*.*

6. $q_{accept}$ *is the* accepting state*.*

7. $q_{reject}$ *is the* rejecting state*.*

We now define how a Turing machine operates. At any moment in time, as explained in the previous section, we consider that a Turing machine is in a state, its memory contents consists of a semi-infinite string over $\Gamma$ and its memory head is scanning one of the memory symbols. This idea is captured by the following formal concept. In the remaining definitions of this section, suppose that $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ is a Turing machine.

**Definition 7.2** *A* configuration *of* $M$ *is a triple* $(q, u, i)$ *where*

1. $q \in Q$ *is the* state *of* $M$*.*

2. $u$ *is a semi-infinite string over* $\Gamma$ *called the* memory contents *of* $M$*.*

3. $i \in \mathbb{Z}^+$ *is the* location *of* $M$*'s* memory head*.*

For example, as explained in the previous section, the *starting configuration* of $M$ on input $w$ is $(q_0, w\textvisiblespace\ldots, 1)$, where "$\textvisiblespace\ldots$" represents an infinite string of blanks.

[1] The term *tape* refers to the magnetic tapes of old computers. Even though it feels outdated, the term nevertheless captures well the sequential nature of a Turing machine's memory.

Here are a few other important configurations. (The starting configuration is repeated for completeness.)

**Definition 7.3** *The* starting configuration *of $M$ on input $w$ is $(q_0, w_\sqcup \ldots, 1)$. A configuration $(q, u, i)$ is* accepting *if $q = q_{accept}$. A configuration $(q, u, i)$ is* rejecting *if $q = q_{reject}$. A configuration is* halting *if it is either accepting or rejecting.*

We are now ready to define what a Turing machine does at every step of its computation.

**Definition 7.4** *Suppose that $(q, u, i)$ is a non-halting configuration of $M$. Suppose that $\delta(q, u_i) = (q', b, D)$. Then,* in one step, *$(q, u, i)$* leads to *the configuration $(q', u', i')$ where*

$$u' = u_1 \ldots u_{i-1} b u_{i+1} \ldots$$

*(that is, $u'$ is $u$ with $u_i$ changed to $b$) and*

$$i' = \begin{cases} i+1 & \text{if } D = R \\ i-1 & \text{if } D = L \text{ and } i > 1 \\ 1 & \text{if } D = L \text{ and } i = 1 \end{cases}$$

Note that this definition only applies to non-halting configurations. In other words, we consider that halting configurations do not lead to other configurations. (That's why they're called *halting*...)

**Definition 7.5** *Let $C_0$ be the starting configuration of $M$ on $w$. Let $C_1, C_2, \ldots$ be the sequence of configurations that $C_0$ leads to. (That is, $C_i$ leads to $C_{i+1}$ in one step, for every $i \geq 0$.) Then $M$* accepts *$w$ if this sequence of configurations is finite and ends in an accepting configuration.*

**Definition 7.6** *The language* recognized *by a Turing machine $M$ (or the* language of $M$*) is the following:*

$$L(M) = \{w \in \Sigma^* \mid w \text{ is accepted by } M\}$$

**Definition 7.7** *A language is* recognizable *(or* Turing-recognizable*) if it is recognized by some Turing machine.*

**Definition 7.8** *A Turing machine* decides *a language if it recognizes the language and halts on every input. A language is* decidable *(or* Turing-decidable*) if it is decided by some Turing machine.*

As mentioned earlier, the Turing machine provides a way to define precisely, mathematically, what we mean by an algorithm:

**Definition 7.9** *An* algorithm *is a Turing machine that halts on every input.*

It is important to note that this is more than just one possible definition. It turns out that every other reasonable notion of algorithm that has ever been proposed has been shown to be equivalent to the Turing machine definition. (We will see some evidence for this later in this chapter.) In other words, the Turing machine appears to be *the only definition possible*. This phenomenon is known as the *Church-Turing Thesis*. So the Turing machine definition of an algorithm is more than just a definition: it can be viewed as a basic law or axiom of computing. And the Church-Turing Thesis has an important consequence: by eliminating the need for competing notions of algorithms, it brings simplicity and clarity to the theory of computation.

## 7.3 Examples

In this section, we start exploring the power of Turing machines by considering some simple examples.

**Example 7.10** Over the alphabet $\{0,1\}$, consider the language of strings of length at least two whose first and last symbols are identical. A Turing machine for this language can be designed based on the following basic idea: look at the first symbol then move the memory head to the last symbol and make sure it is the same.

Here's a more detailed description:

1. If the first symbol is blank, reject. (Because the input is empty.)
2. Remember the first symbol. (By using the states of the machine. More detail later.)
3. Move right.
4. If the current symbol is blank, reject. (Because the input has length 1.)
5. Move right until a blank is found.
6. Move left. (The head is now over the last symbol.)
7. If the current symbol is identical to the first one, accept. Otherwise, reject.

Note how the above description specifies how the Turing machine moves its memory head. But it doesn't specify the states or the transition function of the machine. We call this a **low-level description** of the Turing machine.

Figure 7.2 gives a **full**, or **complete**, **description**. This description takes the form of a state diagram similar to the state diagram of a DFA. All the details on

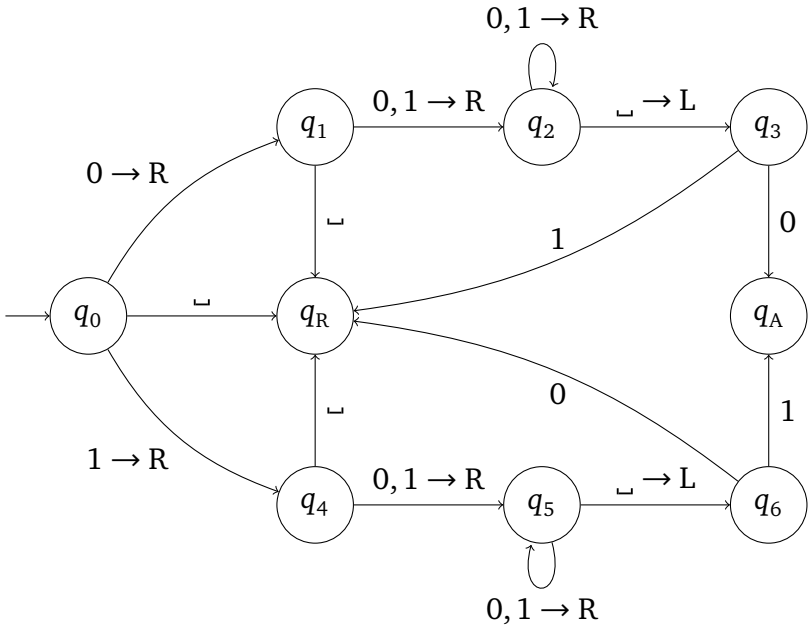



Figure 7.2: A Turing machine for the language of strings of length at least two whose first and last symbols are identical

the states and the transition function of the machine are included in the state diagram. This complete description of the machine is essentially the implementation of the preceding low-level description.

In general, in the state diagram of a Turing machine, a transition labeled $\mathtt{a} \to \mathtt{b}, \mathrm{D}$ from state $q$ to state $r$ means that $\delta(q, \mathtt{a}) = (r, \mathtt{b}, \mathrm{D})$. In other words, that transition should be used if the current symbol is an $\mathtt{a}$, that symbol should be changed to a $\mathtt{b}$ and the head should move in the direction D. In case the symbol is not to be changed, the shorter form $\mathtt{a} \to \mathrm{D}$ may be used as an abbreviation for

$\mathtt{a} \rightarrow \mathtt{a}, \mathrm{D}$. In that case, multiple transitions can also be combined into one, as in $\mathtt{a}, \mathtt{b} \rightarrow \mathrm{D}$. When moving to the accepting or rejecting state, the symbol to be written and the movement of the head are not important (because the machine is about to halt) so we often omit them. The label of the transition is then just a single symbol, as in $\mathtt{a}$, or multiple symbols, as in $\mathtt{a}, \mathtt{b}$. □

This example showed how a Turing machine can remember individual symbols from its input (or memory) as its head moves to other symbols. The technique used is essentially the same one we used with DFA's, which isn't surprising since a Turing machine is essentially a DFA with memory.

The next example revisits the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\}$. We sketched a Turing machine for this language at the end of Section 7.1. We now provide more detail. This will illustrate how a Turing machine can take advantage of its ability to change the contents of the memory.

**Example 7.11** In Section 7.1, we sketched the operation of a Turing machine for the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. That description was missing several details. For example, it said to scan the input from left to right but didn't specify how the Turing machine would detect the end of the input. That turns out to be easy since, initially at least, the input is followed by a blank symbol. We already took advantage of that in the previous example.

But the description of Section 7.1 also said to return the head to the beginning of the memory. This is more difficult since, according to the definitions of the previous section, there is no mechanism to allow a Turing machine to detect the beginning of its memory. A common solution is to change the first symbol of the input to some unique symbol that then becomes a marker for the beginning of the memory. In the case of the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\}$, we can do something simpler: we can use different symbols for crossing off $\mathtt{a}$'s, $\mathtt{b}$'s and $\mathtt{c}$'s. Then,

instead of returning the head to the beginning of the memory, we simply return it to the last a that was crossed off.

Here's a low-level description that uses this idea:

1. If the first symbol is blank, accept. (Because the input is empty.)
2. If the first symbol is not an a, reject. (Because the input has more b's or c's than a's.)
3. Replace the a by an x.
4. Move right, skipping a's and y's, until a b is found. Replace that b with a y. If no b was found, reject.
5. Move right, skipping b's and z's, until a c is found. Replace that c with a z. If no c was found, reject.
6. Move left until an x is found. Move right. (The current symbol is the first a that hasn't been crossed off. Or a y.)
7. Repeat Steps 3 to 6 as long as the current symbol is an a.
8. When that symbol is a y instead, scan the rest of the memory to verify that all b's and c's have been crossed off. This can be done by moving right, skipping y's and z's until a blank symbol is found. If other symbols are found before the first blank, reject. Otherwise, accept.

It is easy to see that this Turing machine will accept every string of the form $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$. To see that it will *only* accept strings of that form, consider what the memory contents could be after the first iteration of the loop. It has to be a string of the form $\mathtt{x}\mathtt{a}^*\mathtt{y}\mathtt{b}^*\mathtt{z}\Sigma^*$. (We're omitting the blanks.) After two iterations, the memory contains a string of the form $\mathtt{x}^2\mathtt{a}^*\mathtt{y}^2\mathtt{b}^*\mathtt{z}^2\Sigma^*$. Suppose that the

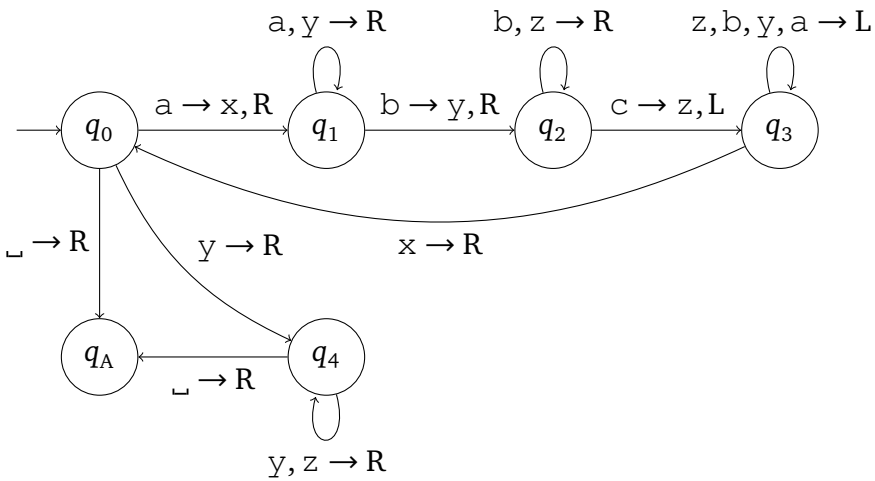



Figure 7.3: A Turing machine for the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$

loop runs for $n$ iterations. At that point, the memory contents is of the form $\mathtt{x}^n\mathtt{y}^n\mathtt{b}^*\mathtt{z}^n\Sigma^*$. The input will then be accepted only if the memory contents is of the form $\mathtt{x}^n\mathtt{y}^n\mathtt{z}^n$, which implies that the input was of the form $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$.

Figure 7.3 gives the transition diagram of a Turing machine that implements the above low-level description. Note that all missing transitions go to the rejecting state. These transitions, as well as the rejecting state itself, are not shown to reduce clutter in the state diagram. □

We could continue giving examples of Turing machines for various languages. But providing low-level or complete descriptions of these machines would quickly become tedious, in large part because these descriptions would end up repeating the implementation of many of the same ideas and techniques. In addition, for more complex languages, low-level and complete descriptions

of Turing machines are completely impractical. Instead, later in this chapter, we will give convincing evidence — but not a formal proof — that Turing machines can simulate computer programs written in typical programming languages. This will allow us to describe Turing machines at a much higher level, using typical pseudocode that includes, for example, variables, loops, conditionals and arrays. We will call this type of description a **high-level description** of a Turing machine.

In the meantime, we end this section by pointing out that every context-free language can be decided by a Turing machine:

**Theorem 7.12** *If a language is context-free, then it is decidable.*

In other words, Turing machines are at least as powerful as context-free grammars and PDA's. But recall in the last example, we showed that the language $\{a^n b^n c^n \mid n \geq 0\}$ is decidable. Since this language is not context-free, we therefore have that Turing machines are more powerful than context-free grammars and PDA's, and that the class of context-free languages is a strict subset of the class of decidable languages.

Theorem 7.12 can be proved by implementing, with a Turing machine, one of the deterministic algorithms for CFL's we discussed in the previous chapter. But this will be much easier to do after we convince ourselves that Turing machines can implement the operations of typical pseudocode.

On the other hand, it is not too difficult to prove right now a special case of Theorem 7.12:

**Theorem 7.13** *If a language is regular, then it is decidable.*

**Proof** The basic idea is simple. Suppose that $M$ is a DFA. A Turing machine can simulate $M$ by reading the input from left to right while going through the same

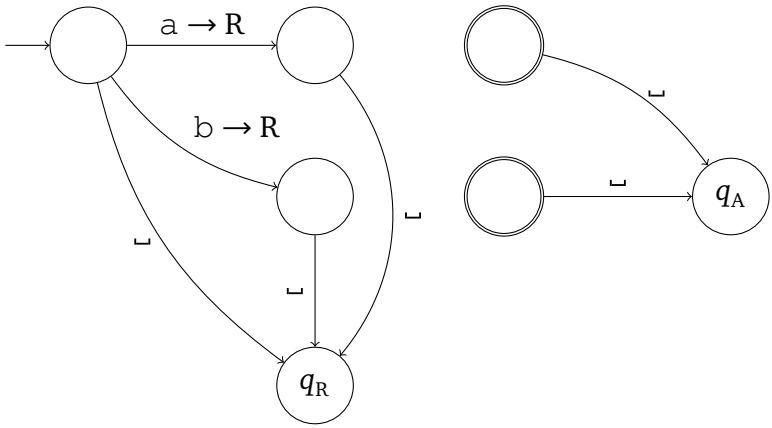



Figure 7.4: A Turing machine that simulates a DFA

states as $M$. Once the end of the input is reached, the Turing machine enters its accepting state if the current state of the DFA is accepting; otherwise, it enters its rejecting state.

To implement this idea, we transform DFA $M$ into Turing machine $M'$ as illustrated in Figure 7.4. The states of $M'$ are the states of the DFA plus an accepting state and a rejecting state. A transition labeled a in the DFA becomes a transition labeled a → R in $M'$. Transitions labeled with the blank symbol are added from every accepting state of the DFA to the accepting state of $M'$ and from every non-accepting state of the DFA to the rejecting state of $M'$.

More precisely, if $M = (Q, \Sigma, \delta, q_0, F)$, then $M' = (Q', \Sigma, \Gamma, \delta', q_0, q_A, q_R)$,

where $Q' = Q \cup \{q_A, q_R\}$, $\Gamma = \Sigma \cup \{\sqcup\}$ and $\delta'$ is defined as follows:

$$\delta'(q,a) = (\delta(q,a), a, \mathrm{R}) \quad \text{if } q \in Q \text{ and } a \neq \sqcup$$

$$\delta'(q,\sqcup) = \begin{cases} (q_A, \sqcup, \mathrm{R}) & \text{if } q \in F \\ (q_R, \sqcup, \mathrm{R}) & \text{if } q \notin F \end{cases}$$

□

The proof of this theorem gives us a general approach for designing Turing machines for regular languages: design a DFA and then convert it into a Turing machine. But note that it is often easier to design the Turing machine directly, as we did in the first example of this section.

### Exercises

7.3.1. Give a low-level and a full description of a Turing machine for the language of strings of length at least two that end in `00`. The alphabet is $\{0, 1\}$.

7.3.2. Give a low-level and a full description of a TM for the language $\{w\#w \mid w \in \{0,1\}^*\}$. The alphabet is $\{0, 1, \#\}$.

7.3.3. Give a low-level description of a TM for the language $\{ww \mid w \in \{0,1\}^*\}$. The alphabet is $\{0, 1\}$.

## 7.4 Variations on the Basic Turing Machine

In the previous section, we defined what we will call our basic Turing machine. In this section, we define two variations and show that they are equivalent to the

basic model. This will constitute some evidence in support of the Church-Turing Thesis.

The first variation is fairly minor. The basic Turing machine must move its head left or right at every move. A *stay option* would allow the Turing machine to not move its head. This can be easily incorporated into the formal definition of the Turing machine by extending the definition of the transition function:

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}$$

The definition of a move must also be extended by adding the following case:

$$i' = i \quad \text{if } D = \mathrm{S}$$

It is clear that a basic Turing machine can be simulated by a Turing machine with the stay option: that option is simply not used.

The reverse is also easy to establish. Suppose that $M$ is a Turing machine with the stay option. A basic Turing machine $M'$ that simulates $M$ can be constructed as follows. For every left and right move, $M'$ operates exactly as $M$. Now suppose that in $M$, $\delta(q_1, a) = (q_2, b, \mathrm{S})$. Then add a new state $q_{1,a}$ to $M'$ and the following transitions:

$$\delta(q_1, a) = (q_{1,a}, b, \mathrm{R})$$

$$\delta(q_{1,a}, c) = (q_2, c, \mathrm{L}), \text{ for every } c \in \Gamma$$

This allows $M'$ to simulate every stay move of $M$ with a right-left combination of moves.

The second variation of the basic Turing machine we will consider in this section is more significant. The basic Turing machine has one memory string. A *multitape* Turing machine is a Turing machine that uses more than one memory

string. Initially, the first memory string contains the input string followed by an infinite number of blanks. The other memory strings are completely blank. The Turing machine has one memory head for each memory string and it can operate each of those heads independently. Note that the number of tapes must be a fixed number, independent of the input string. (A detailed, formal definition of the multitape Turing machine is left as an exercise.)

The basic, single-tape Turing machine is a special case of the multitape Turing machine. Therefore, to establish that the two models are equivalent, all we need to show is that single-tape Turing machines can simulate the multitape version.

**Theorem 7.14** *Every multitape Turing machine has an equivalent single-tape Turing machine.*

**Proof** Suppose that $M_k$ is a $k$-tape Turing machine. We construct a single-tape Turing machine $M_1$ that simulates $M_k$ as follows. Each of the memory symbols of $M_1$ represents $k$ of the memory symbols of $M_k$. For example, if $M_k$ had three tapes containing $x_1x_2x_3\ldots$, $y_1y_2y_3\ldots$ and $z_1z_2z_3\ldots$, respectively, then the single tape of $M_1$ would contain

$$\#(x_1, y_1, z_1)(x_2, y_2, z_2)(x_3, y_3, z_3)\ldots$$

The # symbol at the beginning of the tape will allow $M_1$ to detect the beginning of the memory.

The single-tape machine also needs to keep track of the location of all the memory heads of the multitape machine. This can be done by introducing underlined versions of each tape symbol of $M_k$. For example, if one of the memory heads of $M_k$ is scanning a `1`, then, in the memory of $M_1$, the corresponding `1` will be underlined: $\underline{1}$.

Here is how $M_1$ operates:

1. Initialize the tape so it corresponds to the initial configuration of $M_k$.

2. Scan the tape from left to right to record the $k$ symbols currently being scanned by $M_k$. (The scanning can stop as soon as the $k$ underlined symbols have been seen. $M_1$ remembers those symbols by using its states.)

3. Scan the tape from right to left, updating the scanned symbols and head locations according to the transition function of $M_k$.

4. Repeat Steps 2 and 3 until $M_k$ halts. If $M_k$ accepts, accept. Otherwise, reject.

It should be clear that $M_1$ accepts exactly the same strings as $M_k$. □

## Exercises

7.4.1. Give a formal definition of multitape Turing machines.

7.4.2. The memory string of the basic Turing machine is semi-infinite because it is infinite in only one direction. Say that a memory string is *doubly infinite* if it is infinite in both directions. Show that Turing machines with doubly infinite memory are equivalent to the basic Turing machine.

7.4.3. Show that the class of decidable languages is closed under complementation, union, intersection and concatenation.

7.4.4. Consider the language of strings of the form $x\#y\#z$ where $x$, $y$ and $z$ are strings of digits of the same length that when viewed as numbers, satisfy the equation $x + y = z$. For example, the string `123#047#170` is in this language because $123 + 47 = 170$. The alphabet is $\{0, 1, \ldots, 9, \#\}$. Show that this language is decidable. (Recall that according to Exercise 5.2.3,

this language is not regular. It's also possible to show that it's not context-free.)

7.4.5. A PDA is an NFA augmented by a stack. Let a 2-PDA be a *DFA* augmented by *two* stacks. We know that PDA's recognize exactly the class of CFL's. In contrast, show that 2-PDA's are equivalent to Turing machines.

## 7.5 Equivalence with Programs

As mentioned earlier, the Church-Turing Thesis states that every reasonable notion of algorithm is equivalent to a Turing machine that halts on every input. While it is not possible to prove this statement, there is a lot of evidence in support of it. In fact, every reasonable notion of algorithm that has ever been proposed has been shown to be equivalent to the Turing machine version. We saw two examples of this in the previous section, when we showed that Turing machines with the stay option and multitape Turing machines are equivalent to the basic model.

In this section, we go further and claim that programs written in a high-level programming language are also equivalent to Turing machines. We won't prove this claim in detail because the complexity of high-level programming languages would make such a proof incredibly time-consuming and tedious. But it is possible to get a pretty good sense of why this equivalence holds.

To make things concrete, let's focus on C++. (Or pick your favorite high-level programming language.) First, it should be clear that every Turing machine has an equivalent C++ program. To simulate a Turing machine, a C++ program only needs to keep track of the state, memory contents and memory head location of the machine. At every step, depending on the state and memory symbol currently being scanned, and according to the transition function of the machine,

the program updates the state, memory contents and head location. The simulation starts in the initial configuration of the machine and ends when a halting state is reached.

For the reverse direction, we need to show that every C++ program has an equivalent Turing machine. This is not easy because C++ is a complex language with an enormous number of different features. But we have compilers that can translate C++ programs into assembler. Assuming that we trust that those compilers are correct, then all we need to show is that Turing machines can simulate programs written in assembler. We won't do this in detail, but because assembler languages are fairly simple, it's not hard to see that it could done.

As mentioned in Section 2.1, assembler programs have no variables, no functions, no types and only very simple instructions. An assembler program is a linear sequence of instructions, with no nesting. These instructions directly access data that is stored in memory or in a small set of registers. One of these registers is the program counter, or instruction counter, that keeps track of which instruction is currently being executed. Here are some typical instructions:

- Set the contents of a memory location to a certain value.
- Copy the contents of a memory location to another one.
- Add the contents of a memory location to the contents of another one.
- Jump to another instruction if the contents of a memory location is 0.

To these instructions, we have to add some form of indirect addressing, that is, the fact that a memory location may be used as a pointer that contains the address of another memory location.

To simulate an assembler program, a Turing machine needs to keep track of the contents of the memory and of the various registers. This can be done with

one tape for the memory and one tape for each of the registers. Then, each instruction in the assembler program will be simulated by a group of states within the Turing machine. When the execution of an instruction is over, the machine transitions to the first state of the group that simulates the next instruction.

Because assembler instructions are simple, it's fairly easy to see how a Turing machine can simulate them. For example, consider an instruction that sets the contents of memory location $i$ to the value $x$. Assume that $x$ is a 32-bit number and that the instruction sets the 32 bits that begin at memory location $i$. And note that $i$ and $x$ are actual numbers, not variables. An example of such an instruction would be, set the contents of memory location 17 to the value 63. This can be simulated as follows:

1. Move the memory head to location $i$. (This can be done by scanning the memory from left to right while counting by using the states of the TM.)

2. Write the 32 bits of $x$ to the 32 bits that start at the current memory location.

Now, suppose that we add indirect addressing to this. Here's how we can simulate an instruction that sets to $x$ the contents of the memory location whose address is stored at memory location $i$. We're assuming that memory addresses have 32 bits.

1. Move the memory head to location $i$.

2. Copy the 32 bits that start at the current memory location to an extra tape. Call this value $j$.

3. Scan the memory from left to right. For each symbol, subtract 1 from $j$. When $j$ becomes 1, stop. (The head is now at memory location $j$.)

4. Write the 32 bits of $x$ to the 32 bits that start at the current memory location.

As can be seen from the above, while executing instructions, the Turing machine will need to use a small number of additional tapes to store temporary values. The details of the subtraction in Step 2 are left as an exercise similar to one from the previous section.

Other assembler instructions can be simulated in a similar way. This makes it pretty clear that Turing machines can simulate assembler programs. And, as explained earlier, by combining this with the fact that compilers can translate C++ into assembler, we get that Turing machines can simulate C++ programs.

Up until now, all the descriptions we have given of Turing machines have been either low-level descriptions, such as those in this section, or full descriptions, such as the state diagrams of Figures 7.2 and 7.3. Now that we have convincing evidence that Turing machines can simulate the constructs of high-level programming languages, from now on, we will usually describe Turing machines in pseudocode. We will say that these are **high-level descriptions** of Turing machines. (From now on, unless otherwise indicated, you should assume that exercises ask for high-level descriptions of Turing machines.)

## Exercises

7.5.1. Describe how a Turing machine can simulate each of the following assembler instructions. Give low-level descriptions. In each case, assume that 32 bits are copied, added or tested.

a) Copy the contents of memory location $i$ to memory location $j$.

b) Add the contents of memory location $i$ to the contents of memory location $j$.

c) Jump to another instruction if the contents of a memory location $i$ is 0.

7.5.2. Show that the class of decidable languages is closed under the star operation.

# Chapter 8

# Undecidability

## 8.1 Introduction

In this chapter, we will finally prove that there are computational problems that cannot be solved by any algorithm. More precisely, using the concepts we have studied in these notes, we will prove that there are languages that cannot decided by any Turing machine. We say that these languages are *undecidable*.

One of these languages corresponds to a problem that we mentioned in the first chapter of these notes: the Halting Problem. In general, the input to this problem is an algorithm and the output is the answer to the following question: Does the algorithm halt on every possible input?

As we have learned in these notes, problems like these can be made more precise by formulating them as languages. In this case, we will say that the Halting Problem is the set of Turing machines that halt on every input. But languages are sets of strings, so this requires that every Turing machine be represented by a string.

This can be done in a variety of ways. One way is to simply write out the

formal description of the Turing machine, that is, a description that follows the formal definition of Section 7.2. This requires an alphabet that includes parentheses, commas, braces, letters and digits. One difficulty is that the input or tape alphabets of the Turing machine may include symbols that are not in the alphabet used for the encoding. A solution is to simply number these symbols and rewrite the transition function so it uses those numbers instead of the original symbols. The same can be done for the states in case their original names include symbols that are not in the encoding alphabet.

If $M$ is a Turing machine, let $\langle M \rangle$ denote the encoding of $M$ in whatever encoding scheme we've decided to use. Then the Halting Problem can be defined precisely — formally — as the language of strings of the form $\langle M \rangle$ where $M$ is a TM that halts on every input. Later in this chapter, we will prove that this language is undecidable.

It turns out that pretty much any computational problem concerning Turing machines is undecidable. We will see several examples in this chapter, including the following:

- The *Acceptance Problem*: the language of strings of the form $\langle M, w \rangle$ where $M$ is a TM, $w$ is a string over the input alphabet of $M$, and $M$ accepts $w$.

- The *Emptiness Problem*: the language of strings of the form $\langle M \rangle$ where $M$ is a TM and $L(M)$ is empty. In other words, $M$ accepts no strings.

- The *Equivalence Problem*: the language of strings of the form $\langle M_1, M_2 \rangle$ where $M_1$ and $M_2$ are TM's and $L(M_1) = L(M_2)$. In other words, $M_1$ and $M_2$ accept and reject precisely the same strings.

Before learning how to prove these undecidability results, we will first consider the above problems in the context of finite automata and context-free grammars. In those contexts, most of these problems are actually decidable.

## 8.2 Problems Concerning Finite Automata

The *Acceptance Problem for DFA's* is the language of strings of the form $\langle M, w\rangle$ where $M$ is a DFA, $w$ is a string over the input alphabet of $M$, and $M$ accepts $w$. We denote this language $A_{DFA}$. The corresponding language for Turing machines will be denoted $A_{TM}$.

**Theorem 8.1** *The language $A_{DFA}$ is decidable.*

**Proof** Here's an algorithm:

1. Verify that the input string is of the form $\langle M, w\rangle$ where $M$ is a DFA and $w$ is a string over the input alphabet of $M$. If not, reject.
2. Simulate $M$ on $w$ by using the algorithm of Figure 2.9.
3. If the simulation ends in an accepting state, accept. Otherwise, reject.

□

Note that this immediately implies that every regular language is decidable: if $L$ is a regular language, an algorithm for $L$ would simply simulate a DFA for $L$.

**Corollary 8.2** *Every regular language is decidable.*

We can also define an *Acceptance Problem for NFA's*. This is the language $A_{NFA}$ of strings of the form $\langle N, w\rangle$ where $N$ is an NFA, $w$ is a string over the input alphabet of $N$, and $N$ accepts $w$.

**Theorem 8.3** *The language $A_{NFA}$ is decidable.*

**Proof** The easiest way to show that this problem is decidable is to convert the NFA into a DFA:

1. Verify that the input string is of the form $\langle N, w \rangle$ where $N$ is an NFA and $w$ is a string over the input alphabet of $N$. If not, reject.

2. Convert $N$ into a DFA $M$ by using the algorithm of Section 3.3.

3. Determine if $M$ accepts $w$ by using the algorithm for the acceptance problem for DFA's.

4. Accept if that algorithm accepts. Otherwise, reject.

□

The *Acceptance Problem for Regular Expressions*, $A_{REX}$, can be defined and shown to be decidable in a similar way.

**Theorem 8.4** *The language $A_{REX}$ is decidable.*

**Proof**

1. Verify that the input string is of the form $\langle R, w \rangle$ where $R$ is a regular expression and $w$ is a string over the alphabet of $R$. If not, reject.

2. Convert $R$ into an NFA $N$ by using the algorithm of Section 4.4.

3. Determine if $N$ accepts $w$ by using the algorithm for the acceptance problem for NFA's.

4. Accept if that algorithm accepts. Otherwise, reject.

□

We now turn to a different type of problem: the *Emptiness Problem for DFA's*. This is the language $E_{DFA}$ of strings of the form $\langle M \rangle$ where $M$ is a DFA and $L(M)$ is empty.

**Theorem 8.5** *The language $E_{DFA}$ is decidable.*

**Proof** This problem is equivalent to a graph reachability problem: we want to determine if there is a path from the starting state of $M$ to any of its accepting states. A simple marking algorithm can do the job:

1. Verify that the input string is of the form $\langle M \rangle$ where $M$ is a DFA. If not, reject.

2. Mark the starting state of $M$.

3. Mark any state that can be reached with one transition from a state that's already marked.

4. Repeat Step 3 until no new states get marked.

5. If an accepting state is marked, reject. Otherwise, accept.

This algorithm is a form of breadth-first search. It's correctness is not hard to establish. First, any state marked the first time that Step 3 is executed is reachable from the start state in one step. The second time that step is executed, any state that is marked is reachable from the start state in two steps. And so on. Therefore, every marked state is reachable from the starting state.

The reverse is also true. If a state is reachable from the start state in one step, it will be marked the first time that Step 3 is executed. If a state is reachable from the start state in two steps, then it is connected to a state that is reachable in one step from the start state. Therefore, the state that's reachable in two steps from the start state will be marked the second time that Step 3 is executed. And

so on. Therefore, every state reachable that's from the start state will be marked eventually.[1]

In addition, the algorithm is guaranteed to stop since the DFA has a finite number of states so Step 3 cannot keep marking states forever. □

The *Equivalence Problem for DFA's* is the language $\text{EQ}_{\text{DFA}}$ of strings of the form $\langle M_1, M_2 \rangle$ where $M_1$ and $M_2$ are DFA's and $L(M_1) = L(M_2)$.

**Theorem 8.6** *The language $EQ_{DFA}$ is decidable.*

**Proof** A clever solution to this problem comes from considering the symmetric difference of the two languages:

$$L(M_1) \ominus L(M_2) = (L(M_1) \cap \overline{L(M_2)}) \cup (\overline{L(M_1)} \cap L(M_2))$$

The first key observation is that the symmetric difference is empty if and only if the two languages are equal. The second key observation is that the symmetric difference is regular. That's because the class of regular languages is closed under complementation, intersection and union. In addition, those closure properties are constructive: we have algorithms that can produce DFA's for those languages.

Here's an algorithm that uses all of those ideas:

1. Verify that the input string is of the form $\langle M_1, M_2 \rangle$ where $M_1$ and $M_2$ are DFA's. If not, reject.

[1] The arguments in these two paragraphs can be carried out more formally as proofs by induction. The first argument would be a proof by induction on the number of times that Step 3 is executed. The second one would be by induction on the length of a shortest path from the start state.

2. Construct a DFA $M$ for the language $L(M_1) \ominus L(M_2)$ by using the algorithms of Section 2.5.

3. Test if $L(M) = \emptyset$ by using the emptiness algorithm.

4. Accept if that algorithm accepts. Otherwise, reject.

□

### Exercises

8.2.1. Let $\text{ALL}_{\text{DFA}}$ be the language of strings of the form $\langle M \rangle$ where $M$ is a DFA that accepts every possible string over its input alphabet. Show that $\text{ALL}_{\text{DFA}}$ is decidable.

8.2.2. Let $\text{SUBSET}_{\text{REX}}$ be the language of strings of the form $\langle R_1, R_2 \rangle$ where $R_1$ and $R_2$ are regular expressions and $L(R_1) \subseteq L(R_2)$. Show that $\text{SUBSET}_{\text{REX}}$ is decidable.

8.2.3. Consider the language of strings of the form $\langle M \rangle$ where $M$ is a DFA that accepts at least one string of odd length. Show that this language is decidable.

## 8.3 Problems Concerning Context-Free Grammars

The *Acceptance Problem for CFG's* is the language $\text{A}_{\text{CFG}}$ of strings of the form $\langle G, w \rangle$ where $G$ is a CFG, $w$ is a string of terminals, and $G$ derives $w$.

**Theorem 8.7** *The language $A_{CFG}$ is decidable.*

**Proof** An algorithm for $\text{A}_{\text{CFG}}$ can be designed in a straightforward way:

1. Verify that the input string is of the form $\langle G, w \rangle$ where $G$ is a CFG and $w$ is a string of terminals. If not, reject.

2. Convert $G$ into an equivalent CFG $G'$ in Chomsky Normal Form, by using the algorithm mentioned in Section 6.9.

3. Determine if $G'$ derives $w$ by using the CFL algorithm of Section 6.9.

4. Accept if that algorithm accepts. Otherwise, reject.

□

The algorithms of Section 6.9 were not described in detail but this can be done and it can be shown that these algorithms can be implemented by Turing machines.

Note that the above theorem implies that every CFL is decidable.

**Corollary 8.8** *Every CFL is decidable.*

The *Emptiness Problem for CFG's* is the language $\mathrm{E}_{\mathrm{CFG}}$ of strings of the form $\langle G \rangle$ where $G$ is a CFG and $L(G)$ is empty.

**Theorem 8.9** *The language $E_{CFG}$ is decidable.*

**Proof** Just like the Emptiness Problem for DFA's, the Emptiness Problem for CFG's can be solved by a simple marking algorithm. The idea is to mark variables that can derive at least one string of terminals.

1. Verify that the input string is of the form $\langle G \rangle$ where $G$ is a CFG. If not, reject.

2. Mark all the terminals of $G$.

3. Mark any variable $A$ for which $G$ has a rule $A \to U_1 \cdots U_k$ where each $U_i$ has already been marked.

4. Repeat Step 3 until no new variables get marked.

5. If the start variable of $G$ is marked, reject. Otherwise, accept.

The correctness of this algorithm is not hard to establish. First, it is clear that if a variable is marked, then that variable can derive a string of terminals. Second, it's not hard to show that if a variable can derive a string of terminals with a parse tree of height $h$, then that variable will be marked after no more than $h$ iterations of the loop in the algorithm.[2] □

What about the *Equivalence Problem for CFG's*? This is the language $\text{EQ}_{\text{CFG}}$ of strings of the form $\langle G_1, G_2 \rangle$ where $G_1$ and $G_2$ are CFG's and $L(G_1) = L(G_2)$. The strategy we used in the previous section for DFA's does not work here because the class of CFL's is not closed under complementation or intersection. In fact, it turns out that $\text{EQ}_{\text{CFG}}$ is undecidable: there is no Turing machine that can decide this language. Note that by the Church-Turing Thesis, this means that there is no algorithm of any kind that can decide this language. We will learn how to prove undecidability results in the following sections.

## Exercises

8.3.1. Let $\text{DERIVES}\varepsilon_{\text{CFG}}$ be the language of strings of the form $\langle G \rangle$ where $G$ is a CFG that derives $\varepsilon$. Show that $\text{DERIVES}\varepsilon_{\text{CFG}}$ is decidable.

[2] Once again, these arguments can be carried out more formally as proofs by induction. The first argument would be a proof by induction on the number of times that Step 3 is executed. The second one would be by in induction on the height of a parse tree that derives a string of terminals.

8.3.2. Consider the language of strings of the form $\langle G \rangle$ where $G$ is a CFG that accepts at least one string of odd length. Show that this language is decidable.

8.3.3. Let $\text{INFINITE}_{\text{CFG}}$ be the language of strings of the form $\langle G \rangle$ where $G$ is a CFG that derives an infinite number of strings. Show that $\text{INFINITE}_{\text{CFG}}$ is decidable. *Hint*: Use the Pumping Lemma.

## 8.4 An Unrecognizable Language

In this section, we show that for every alphabet $\Sigma$, there is at least one language that cannot recognized by any Turing machine. The proof of this result uses the diagonalization technique.

**Theorem 8.10** *Over every alphabet $\Sigma$, there is a language that is not recognizable.*

**Proof** Let $s_1, s_2, s_3, \ldots$ be a lexicographic listing of all the strings in $\Sigma^*$.[3] Let $M_1, M_2, M_3, \ldots$ be a lexicographic listing of all the Turing machines with input alphabet $\Sigma$. This list can be obtained from a lexicographic listing of all the valid encodings of Turing machines.

The languages of all these machines can be described by a table with a row for each machine and a column for each string. The entry for machine $M_i$ and string $s_j$ indicates whether $M_i$ accepts $s_j$. Figure 8.1 shows an example where $A$ indicates acceptance and $N$ indicates nonacceptance.

[3] In common usage, the adjective *lexicographic* relates to dictionaries, or their making. In the context of formal languages, however, the alphabetical ordering used in dictionaries is problematic. For example, an alphabetical listing of all strings over the alphabet $\{0, 1\}$ would be $\varepsilon$, 0, 00, 000,... Strings with 1's wouldn't get listed. Instead, in a *lexicographic ordering*, smaller strings are listed first. Over the alphabet $\{0, 1\}$, for example, this gives $\varepsilon$, 0, 1, 00, 01, 10, 11, 000, 001, 010,...

| | $s_1$ | $s_2$ | $s_3$ | $\cdots$ |
|---|---|---|---|---|
| $M_1$ | A | N | A | $\cdots$ |
| $M_2$ | A | A | N | $\cdots$ |
| $M_3$ | N | A | N | $\cdots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | |

Figure 8.1: The table of acceptance for all Turing machines over $\Sigma$

Our objective is to find a language $L$ that cannot be recognized by any of these Turing machines. In other words, we have to guarantee that for each Turing machine $M_i$, there is are least one input string $s_j$ on which $M_i$ and the language disagree. That is, a string $s_j$ that is contained in $L$ but rejected by $M_i$, or that is not contained in $L$ but accepted by $M_i$.

To achieve this, note that the diagonal of the above table defines the following language:

$$D = \{s_i \mid M_i \text{ accepts } s_i\}$$

Now consider the complement of $D$:

$$\overline{D} = \{s_i \mid M_i \text{ does not accept } s_i\}$$

This language is defined by the *opposite* of the entries on the diagonal. We claim that $\overline{D}$ is not recognizable.

Suppose, for the sake of contradiction, that $\overline{D}$ is recognizable. Then it must be recognized by one of these machines. Suppose that $\overline{D} = L(M_i)$. Now, consider what $M_i$ does when it runs on $s_i$.

If $M_i$ accepts $s_i$, then, by the definition of $\overline{D}$, $s_i \notin \overline{D}$. But since $M_i$ recognizes

$\overline{D}$, it must be that $M_i$ does *not* accept $s_i$. That's a contradiction.

On the other hand, if $M_i$ does not accept $s_i$, then the definition of $\overline{D}$ implies that $s_i \in \overline{D}$. Once again, since $M_i$ recognizes $\overline{D}$, it must be that $M_i$ accepts $s_i$. That's also a contradiction.

Therefore, in either case, we get a contradiction. This implies that $\overline{D}$ is not recognizable. □

Note that since $\overline{D}$ is not recognizable, it is of course also undecidable. But this also implies that $D$ is undecidable since the class of decidable languages is closed under complementation.

### Exercises

8.4.1. Suppose that $\Sigma$ is an alphabet over which an encoding of Turing machines can be defined.[4] Consider the language $L$ of strings of the form $\langle M \rangle$ where $M$ is a Turing machine with input alphabet $\Sigma$ and $M$ does not accept the string $\langle M \rangle$. Show that the language $L$ is not recognizable.

## 8.5 Natural Undecidable Languages

In the previous section, we showed that unrecognizable languages exist. But the particular languages we considered were somewhat artificial, in the sense that they were constructed specifically for the purposes of proving this result. In this section, we consider two languages that are arguably more natural and show that they are undecidable.

[4] It turns out that this can be any alphabet. It's not hard to see that Turing machines can be described in every alphabet of size at least 2. It's a little trickier to show that it can also be done over alphabets of size 1.

The first language is the *Acceptance Problem for Turing Machines*. This is the language $A_{TM}$ of strings of the form $\langle M, w\rangle$ where $M$ is a Turing machine, $w$ is a string over the input alphabet of $M$ and $M$ accepts $w$.

**Theorem 8.11** *The language $A_{TM}$ is undecidable.*

**Proof** By contradiction. Suppose that algorithm $R$ decides $A_{TM}$. We use this algorithm to design an algorithm $S$ for the language $\overline{D}$ defined in the proof of Theorem 8.10. Recall that

$$\overline{D} = \{s_i \mid M_i \text{ does not accept } s_i\}$$

Let $\Sigma$ be the alphabet of $A_{TM}$. Here's a description of $S$:

1. Let $w$ be the input string.
2. Find $i$ such that $w = s_i$. (This can be done by listing all the strings $s_1, s_2, s_3, \ldots$ until $w$ is found.)
3. Generate the encoding of machine $M_i$. (This can be done by listing all the valid encodings of Turing machines.)
4. Run $R$ on $\langle M_i, s_i\rangle$.
5. If $R$ accepts, reject. Otherwise, accept.

It is easy to see that $S$ decides $\overline{D}$. Since this language is not even recognizable, this is a contradiction. This shows that $R$ cannot exist and that $A_{TM}$ is undecidable. □

The second language we consider in this section is a version of the famous Halting Problem we mentioned in the introduction of these notes and the introduction of this chapter. Consider the language $HALT_{TM}$ of strings of the form

$\langle M, w \rangle$ where $M$ is a Turing machine, $w$ is a string over the input alphabet of $M$ and $M$ halts on $w$.[5]

**Theorem 8.12** *The language $HALT_{TM}$ is undecidable.*

**Proof** By contradiction. Suppose that algorithm $R$ decides $\text{HALT}_{\text{TM}}$. We use this algorithm to design an algorithm $S$ for the Acceptance Problem:

1. Verify that the input string is of the form $\langle M, w \rangle$ where $M$ is a Turing machine and $w$ is a string over the input alphabet of $M$. If not, reject.

2. Run $R$ on $\langle M, w \rangle$.

3. If $R$ rejects, reject. Otherwise, simulate $M$ on $w$.

4. If $M$ accepts, accept. Otherwise, reject.

It is easy to see that $S$ decides $\text{A}_{\text{TM}}$. Since this language is undecidable, this is a contradiction. Therefore, $R$ cannot exist and $\text{HALT}_{\text{TM}}$ is undecidable. □

## Exercises

8.5.1. Show that the language $D$ defined in the proof of Theorem 8.10 is recognizable.

[5]The version of the Halting Problem mentioned earlier took only a machine as input and the problem was to determine if that machine halted on *every* input. This corresponds to the language $\text{HALTS\_ON\_ALL}_{\text{TM}}$ of strings of the form $\langle M \rangle$ where $M$ is a Turing machine that halts on every input string. The undecidability of $\text{HALTS\_ON\_ALL}_{\text{TM}}$ will be left as an exercise for the next section.

## 8.6 Reducibility and Additional Examples

The two undecidability proofs of the previous section followed the same pattern, which goes like this. To show that a language $B$ is undecidable, start by assuming the opposite, that there is an algorithm $R$ that decides $B$. Then use $R$ to design an algorithm $S$ that decides a language $A$ that is known to be undecidable. This is a contradiction, which shows that $R$ cannot exist and that $B$ is undecidable.

When an algorithm for $A$ can be constructed from an algorithm for $B$, we say that $A$ *reduces* to $B$. Reductions can be used in two ways.

1. To prove decidability results: if $B$ is decidable, then a reduction of $A$ to $B$ shows that $A$, too, is decidable.

2. To prove undecidability results: if $A$ is undecidable, then a reduction of $A$ to $B$ shows that $B$, too, is undecidable (since otherwise $A$ would be decidable, a contradiction).

In the previous section, to prove that $\text{A}_{\text{TM}}$ is undecidable, we reduced $\overline{D}$ to $\text{A}_{\text{TM}}$. To prove that $\text{HALT}_{\text{TM}}$ is undecidable, we reduced $\text{A}_{\text{TM}}$ to $\text{HALT}_{\text{TM}}$. In this section, we will prove four additional undecidability results. In all cases, we will use the reducibility technique.

Consider the language $\text{WRITES\_ON\_TAPE}_{\text{TM}}$ of strings of the form $\langle M, w, a\rangle$ where $M$ is a Turing machine, $w$ is a string over the input alphabet of $M$, $a$ is a symbol in the tape alphabet of $M$, and $M$ writes an $a$ on its tape while running on $w$.

**Theorem 8.13** *The language $\text{WRITES\_ON\_TAPE}_{TM}$ is undecidable.*

**Proof** By contradiction. Suppose that algorithm $R$ decides the language $\text{WRITES\_ON\_TAPE}_{\text{TM}}$. We use this algorithm to design an algorithm $S$ for the Acceptance Problem.

The idea behind the design of $S$ is that when given a Turing machine $M$ and an input string $w$, $S$ transforms $M$ to make it write an a on its tape if, and only if, it's about to accept $w$.

Here's the description of $S$. In this description, the transformed $M$ is called $M'$.

1. Verify that the input string is of the form $\langle M, w \rangle$ where $M$ is a Turing machine and $w$ is a string over the input alphabet of $M$. If not, reject.

2. Let a be a new symbol not in the tape alphabet of $M$. Construct the following Turing machine $M'$:

   (a) Let $x$ be the input string. Run $M$ on $x$.

   (b) If $M$ accepts, write an a on the tape and accept. Otherwise, reject.

3. Run $R$ on $\langle M', w, \mathtt{a} \rangle$.

4. If $R$ accepts, accept. Otherwise, reject.

It's easy to see that $S$ decides $A_{TM}$. First, suppose that $M$ accepts $w$. Then, when $M'$ runs on $w$, it finds that $M$ accepts $w$ and writes an a on its tape. This implies that $R$ accepts $\langle M', w, \mathtt{a} \rangle$ and that $S$ accepts $\langle M, w \rangle$, which is what we want.

Second, suppose that $M$ does not accept $w$. Then when $M'$ runs on $w$, either it finds that $M$ rejects $w$ or it gets stuck simulating $M$ forever. In either case, $M'$ doesn't write an a on its tape. This implies that $R$ rejects $\langle M', w, \mathtt{a} \rangle$ and that $S$ rejects $\langle M, w \rangle$. Therefore, $S$ decides $A_{TM}$.

However, since $A_{TM}$ is undecidable, this is a contradiction. Therefore, $R$ does not exist and WRITES_ON_TAPE$_{TM}$ is undecidable. □

The emptiness problem for Turing machines is the language $E_{TM}$ of strings of the form $\langle M \rangle$ where $M$ is a Turing machine and $L(M) = \emptyset$.

**Theorem 8.14** *The language $E_{TM}$ is undecidable.*

**Proof** By contradiction. Suppose that algorithm $R$ decides $E_{TM}$. We use this algorithm to design an algorithm $S$ for the Acceptance Problem. The idea is that given a Turing machine $M$ and a string $w$, $S$ constructs a new Turing machine $M'$ whose language is empty if and only if $M$ does not accept $w$.

1. Verify that the input string is of the form $\langle M, w \rangle$ where $M$ is a Turing machine and $w$ is a string over the input alphabet of $M$. If not, reject.

2. Construct the following Turing machine $M'$:

   (a) Let $x$ be the input string.

   (b) If $x \neq w$, reject.

   (c) Run $M$ on $w$.

   (d) If $M$ accepts, accept. Otherwise, reject.

3. Run $R$ on $\langle M' \rangle$.

4. If $R$ accepts, reject. Otherwise, accept.

To prove that $S$ decides $A_{TM}$, first suppose that $M$ accepts $w$. Then $L(M') = \{w\}$, which implies that $R$ rejects $\langle M' \rangle$ and that $S$ accepts $\langle M, w \rangle$. On the other hand, suppose that $M$ does not accept $w$. Then $L(M') = \emptyset$, which implies that $R$ accepts $\langle M' \rangle$ and that $S$ rejects $\langle M, w \rangle$.

Since $A_{TM}$ is undecidable, this is a contradiction. Therefore, $R$ does not exist and $E_{TM}$ is undecidable. □

Here's a different version of this proof. The main difference in the definition of Turing machine $M'$.

**Proof** By contradiction. Suppose that algorithm $R$ decides $E_{TM}$. We use this algorithm to design an algorithm $S$ for the Acceptance Problem. The idea is the same as in the previous proof: given a Turing machine $M$ and a string $w$, $S$ constructs a new Turing machine $M'$ whose language is empty if and only if $M$ does not accept $w$.

1. Verify that the input string is of the form $\langle M, w\rangle$ where $M$ is a Turing machine and $w$ is a string over the input alphabet of $M$. If not, reject.

2. Construct the following Turing machine $M'$:

   (a) Let $x$ be the input string.

   (b) Run $M$ on $w$.

   (c) If $M$ accepts, accept. Otherwise, reject.

3. Run $R$ on $\langle M'\rangle$.

4. If $R$ accepts, reject. Otherwise, accept.

To prove that $S$ decides $A_{TM}$, first suppose that $M$ accepts $w$. Then $L(M') = \Sigma^*$, where $\Sigma$ is the input alphabet of $M'$, which implies that $R$ rejects $\langle M'\rangle$ and that $S$ accepts $\langle M, w\rangle$. On the other hand, suppose that $M$ does not accept $w$. Then $L(M') = \emptyset$, which implies that $R$ accepts $\langle M'\rangle$ and that $S$ rejects $\langle M, w\rangle$.

Since $A_{TM}$ is undecidable, this is a contradiction. Therefore, $R$ does not exist and $E_{TM}$ is undecidable. □

The equivalence problem for Turing machines is the language $EQ_{TM}$ of strings of the form $\langle M_1, M_2\rangle$ where $M_1$ and $M_2$ are Turing machines and $L(M_1) = L(M_2)$.

**Theorem 8.15** *The language $EQ_{TM}$ is undecidable.*

**Proof** This time, we will do a reduction from a language other than $A_{TM}$. Suppose that algorithm $R$ decides $EQ_{TM}$. We use this algorithm to design an algorithm $S$ for the Emptiness Problem:

1. Verify that the input string is of the form $\langle M \rangle$ where $M$ is a Turing machine. If not, reject.
2. Construct a Turing machine $M'$ that rejects every input.
3. Run $R$ on $\langle M, M' \rangle$.
4. If $R$ accepts, accept. Otherwise, reject.

This algorithm accepts $\langle M \rangle$ if and only if $L(M) = L(M') = \emptyset$. Therefore, $S$ decides $E_{TM}$, which is contradicts the fact that $E_{TM}$ is undecidable. This proves that $R$ does not exist and that $EQ_{TM}$ is undecidable. □

The *Regularity Problem for Turing Machines* is to determine if the language of a given Turing machine is regular. This corresponds to the language $REGULAR_{TM}$ of strings of the form $\langle M \rangle$ where $M$ is a Turing machine and $L(M)$ is regular.

**Theorem 8.16** *The language $REGULAR_{TM}$ is undecidable.*

**Proof** By contradiction. Suppose that algorithm $R$ decides $REGULAR_{TM}$. We use this algorithm to design an algorithm $S$ for the Acceptance Problem. Given a Turing machine $M$ and a string $w$, $S$ constructs a new Turing machine $M'$ whose language is regular if and only if $M$ accepts $w$.

1. Verify that the input string is of the form $\langle M, w \rangle$ where $M$ is a Turing machine and $w$ is a string over the input alphabet of $M$. If not, reject.

2. Construct the following Turing machine $M'$ with input alphabet $\{0,1\}$:
   (a) Let $x$ be the input string.
   (b) If $x \in \{0^n1^n \mid n \geq 0\}$, accept.
   (c) Otherwise, run $M$ on $w$.
   (d) If $M$ accepts, accept. Otherwise, reject.
3. Run $R$ on $\langle M' \rangle$.
4. If $R$ accepts, accept. Otherwise, reject.

To prove that $S$ decides $A_{TM}$, first suppose that $M$ accepts $w$. Then $L(M') = \{0,1\}^*$, which implies that $R$ accepts $\langle M' \rangle$ and that $S$ accepts $\langle M, w \rangle$. On the other hand, suppose that $M$ does not accept $w$. Then $L(M') = \{0^n1^n \mid n \geq 0\}$, which implies that $R$ rejects $\langle M' \rangle$ and that $S$ rejects $\langle M, w \rangle$. Therefore, $S$ decides $A_{TM}$.

Since $A_{TM}$ is undecidable, this is a contradiction. Therefore, $R$ does not exist and $REGULAR_{TM}$ is undecidable. □

## Exercises

8.6.1. Consider the problem of detecting if a Turing machine $M$ ever attempts to move left from the first position of its tape while running on $w$. This corresponds to the language $BUMPS\_OFF\_LEFT_{TM}$ of strings of the form $\langle M, w \rangle$ where $M$ is a Turing machine, $w$ is a string over the input alphabet of $M$ and $M$ attempts to move left from the first position of its tape while running on $w$. Show that $BUMPS\_OFF\_LEFT_{TM}$ is undecidable.

8.6.2. Consider the problem of determining if a Turing machine $M$ ever enters state $q$ while running on $w$. This corresponds to the language

ENTERS_STATE$_{TM}$ of strings of the form $\langle M, w, q\rangle$ where $M$ is a Turing machine, $w$ is a string over the input alphabet of $M$, $q$ is a state of $M$ and $M$ enters $q$ while running on $w$. Show that ENTERS_STATE$_{TM}$ is undecidable.

8.6.3. Consider the problem of determining if a Turing machine $M$ ever changes the contents of memory location $i$ while running on an input $w$. This corresponds to the language of strings of the form $\langle M, w, i\rangle$ where $M$ is a Turing machine, $w$ is a string over the input alphabet of $M$, $i$ is a positive integer and $M$ changes the value of the symbol at position $i$ of the memory while running on $w$. Show that this language is undecidable.

8.6.4. Let ACCEPTS$\varepsilon_{TM}$ be the language of strings of the form $\langle M\rangle$ where $M$ is a Turing machine that accepts the empty string. Show that ACCEPTS$\varepsilon_{TM}$ is undecidable.

8.6.5. Let HALTS_ON_ALL$_{TM}$ be the language of strings of the form $\langle M\rangle$ where $M$ is a Turing machine that halts on every input string. Show that HALTS_ON_ALL$_{TM}$ is undecidable. *Hint*: Reduce HALT$_{TM}$ to HALTS_ON_ALL$_{TM}$.

8.6.6. Let ALL$_{CFG}$ be the language of strings of the form $\langle G\rangle$ where $G$ is a CFG over some alphabet $\Sigma$ and $L(G) = \Sigma^*$. It is possible to show that ALL$_{CFG}$ is undecidable. Use this result to show that EQ$_{CFG}$ is undecidable.

8.6.7. Let INFINITE$_{TM}$ be the language of strings of the form $\langle M\rangle$ where $M$ is a Turing machine and $L(M)$ is infinite. Show that INFINITE$_{TM}$ is undecidable.

8.6.8. In the proof of Theorem 8.16, modify the construction of $M'$ as follows:

2. Construct the following Turing machine $M'$ with input alphabet $\{0,1\}$:

(a) Let $x$ be the input string.

(b) If $x \notin \{0^n1^n \mid n \geq 0\}$, reject.

(c) Otherwise, run $M$ on $w$.

(d) If $M$ accepts, accept. Otherwise, reject.

Can the proof of Theorem 8.16 be made to work with this alternate construction of $M'$? Explain.

8.6.9. Let $\text{DECIDABLE}_{\text{TM}}$ be the language of strings of the form $\langle M \rangle$ where $M$ is a Turing machine and $L(M)$ is decidable. Show that $\text{DECIDABLE}_{\text{TM}}$ is undecidable.

## 8.7 Rice's Theorem

In the previous section, we showed that the Emptiness and Regularity problems for Turing machines are undecidable. Formally, these problems correspond to the languages $\text{E}_{\text{TM}}$ and $\text{REGULAR}_{\text{TM}}$. In addition, in the exercises of that section, you were asked to show that $\text{ACCEPTS}\varepsilon_{\text{TM}}$, $\text{INFINITE}_{\text{TM}}$ and $\text{DECIDABLE}_{\text{TM}}$ are also undecidable.

These undecidability proofs have a lot in common. And this comes, in part, from the fact that these problems are of a common type: they each involve determining if the language of a Turing machine satisfies a certain property, such as being empty, regular or containing the empty string.

It turns out that we can exploit this commonality to generalize these undecidability results. First, we need to define precisely the general concepts we're dealing with.

**Definition 8.17** *A* property of recognizable languages *is a unary predicate on the set of recognizable languages.*

In other words, a property of recognizable languages is a Boolean function whose domain is the set of recognizable languages. That is, a function that is true for some recognizable languages, and false for the others. For example, the emptiness property is the function that is true for the empty language and false for all other recognizable languages.

Now, to each property $P$ of recognizable languages, we can associate a language: the language $L_P$ of Turing machines whose language satisfies property $P$:

$$L_P = \{\langle M \rangle \mid M \text{ is a TM and } P(L(M)) \text{ is true}\}$$

All of the undecidable languages we mentioned at the beginning of this section are of this form. For example, $\mathrm{E_{TM}} = L_P$ where $P$ is the property of being empty.

The general result we are going to prove in this section is that every language of this form is undecidable. That is, if $P$ is any property of recognizable languages, then $L_P$ undecidable.

Now, there are two obvious exceptions to this result: the two trivial properties, the one that is true for all recognizable languages, and the one that is false for all recognizable languages. These properties can be decided by algorithms that always accept or always reject every Turing machine. So our general result will only apply to non-trivial properties of recognizable languages.

**Definition 8.18** *A property $P$ of recognizable languages is* trivial *if it is true for all recognizable languages or if it is false for all recognizable languages.*

When a language of the form $L_P$ is undecidable, we say that the property $P$ itself is undecidable.

**Definition 8.19** *A property $P$ of recognizable languages is* undecidable *if the language $L_P$ is undecidable.*

We're now ready to state the main result of this section:

**Theorem 8.20 (Rice's Theorem)** *Every non-trivial property of recognizable languages is undecidable.*

**Proof** Suppose that $P$ is a non-trivial property of recognizable languages. First assume that the empty language satisfies property $P$. That is, $P(\emptyset)$ is true. Since $P$ is non-trivial, there is a recognizable language $L$ that doesn't satisfy $P$. Let $M_L$ be a Turing machine that recognizes $L$.

Now, by contradiction, suppose that algorithm $R$ decides $L_P$. Here's how we can use this algorithm to design an algorithm $S$ for the acceptance problem. Given a Turing machine $M$ and a string $w$, $S$ constructs a new Turing machine $M'$ whose language whose language is either empty or $L$ depending on whether $M$ accepts $w$.

1. Verify that the input string is of the form $\langle M, w\rangle$ where $M$ is a Turing machine and $w$ is a string over the input alphabet of $M$. If not, reject.

2. Construct the following Turing machine $M'$. The input alphabet of $M'$ is the same as that of $M_L$.

   (a) Let $x$ be the input string and run $M_L$ on $x$.

   (b) If $M_L$ rejects, reject.

   (c) Otherwise (if $M_L$ accepts), run $M$ on $w$.

   (d) If $M$ accepts, accept. Otherwise, reject.

3. Run $R$ on $\langle M'\rangle$.

4. If $R$ accepts, reject. Otherwise, accept.

To show that $S$ correctly decides the acceptance problem, suppose first that $M$ accepts $w$. Then $L(M') = L$. Since $L$ does not have property $P$, $R$ rejects $\langle M' \rangle$ and $S$ correctly accepts $\langle M, w \rangle$.

On the other hand, suppose that $M$ does not accept $w$. Then $L(M') = \emptyset$, which does have property $P$. Therefore, $R$ accepts $\langle M' \rangle$ and $S$ correctly rejects $\langle M, w \rangle$.

This shows that $S$ decides $A_{TM}$. Since this language is undecidable, this proves that $R$ does not exist and that $L_P$ is undecidable.

Recall that this was all done under the assumption that the empty language has property $P$. Suppose now that the empty language doesn't have property $P$. In that case, modify the above argument as follows. First, let $L$ be a language that *has* property $P$. Second, modify Step 4 of $S$ as follows:

4. If $R$ accepts, *accept*. Otherwise, *reject*.

It is easy to verify that this modified $S$ decides $A_{TM}$ so the conclusion is the same: $L_P$ is undecidable. □

## Exercises

8.7.1. In the proof of Rice's Theorem, modify the construction of $M'$ as follows:

2. Construct the following Turing machine $M'$. The input alphabet of $M'$ is the same as that of $M_L$.
   (a) Let $x$ be the input string. Run $M$ on $w$.
   (b) If $M$ rejects, reject.
   (c) Otherwise (if $M$ accepts), run $M_L$ on $x$.
   (d) If $M_L$ accepts, accept. Otherwise, reject.

Does the proof of Rice's Theorem work with this alternate construction of $M'$? Explain.

## 8.8 Natural Unrecognizable Languages

We have now seen several examples of natural undecidable languages. But we only have one example of an unrecognizable language: the complement of the diagonal language $D$. In this section, we will learn a technique for proving that certain more natural languages are unrecognizable.

What language would be a candidate? A first idea would be the acceptance problem. But it turns out that $A_{TM}$ is recognizable:

1. Verify that the input string is of the form $\langle M, w \rangle$ where $M$ is a Turing machine and $w$ is a string over the input alphabet of $M$. If not, reject.

2. Simulate $M$ on $w$.

3. If $M$ accepts, accept. Otherwise, reject.

This Turing machine is called the *Universal Turing Machine*. It is essentially a Turing machine interpreter.

What about the complement of the acceptance problem? It turns out that if $\overline{A}_{TM}$ was recognizable, then $A_{TM}$ would be decidable, which we know is not the case.

**Theorem 8.21** *The language $\overline{A}_{TM}$ is not recognizable.*

**Proof** Suppose that $\overline{A}_{TM}$ is recognizable. Let $R$ be a Turing machine that recognizes this language. Let $U$ be the Universal Turing Machine. We use both $R$ and $U$ to design an algorithm $S$ for the acceptance problem:

1. Verify that the input string is of the form $\langle M, w \rangle$ where $M$ is a Turing machine and $w$ is a string over the input alphabet of $M$. If not, reject.

2. Run $R$ and $U$ at the same time on $\langle M, w \rangle$. (This can be done by alternating between $R$ and $U$, one step at a time.)

3. If $R$ accepts, reject. If $R$ rejects, accept. If $U$ accepts, accept. If $U$ rejects, reject.

If $M$ accepts $w$, then $U$ is guaranteed to accept. It is possible that $R$ rejects first. In either case, $S$ accepts $\langle M, w \rangle$. Similarly, if $M$ does not accept $w$, then $R$ is guaranteed to accept. It is possible that $U$ rejects first. In either case, $S$ rejects $\langle M, w \rangle$. Therefore, $S$ decides $A_{TM}$.

This contradicts the fact that $A_{TM}$ is undecidable. Therefore, $R$ cannot exist and $\overline{A}_{TM}$ is not recognizable. □

The technique that was used in this proof can be generalized and used to prove other unrecognizabibility results.

**Theorem 8.22** *Let $L$ be any language. If both $L$ and $\overline{L}$ are recognizable, then $L$ and $\overline{L}$ are decidable.*

**Proof** Suppose that $L$ and $\overline{L}$ are both recognizable. Let $R_1$ and $R_2$ be Turing machines that recognize $L$ and $\overline{L}$, respectively. We use $R_1$ and $R_2$ to design an algorithm $S$ that *decides* $L$:

1. Let $w$ be the input string. Run $R_1$ and $R_2$ at the same time on $w$. (This can be done by alternating between $R_1$ and $R_2$, one step at a time.)

2. If $R_1$ accepts, accept. If $R_1$ rejects, reject. If $R_2$ accepts, reject. If $R_2$ rejects, accept.

It is easy to show that $S$ decides $L$. Since the class of decidable languages is closed under complementation, we get that both $L$ and $\overline{L}$ are decidable. □

**Corollary 8.23** *If $L$ is recognizable but undecidable, then $\overline{L}$ is unrecognizable.*

Let's apply this to the halting problem. It is easy to see that $\text{HALT}_{\text{TM}}$ is recognizable:

1. Verify that the input string is of the form $\langle M, w\rangle$ where $M$ is a Turing machine and $w$ is a string over the input alphabet of $M$. If not, reject.

2. Simulate $M$ on $w$.

3. If $M$ halts, accept.

Therefore, by the corollary, $\overline{\text{HALT}_{\text{TM}}}$ is unrecognizable.

Let's now consider the emptiness problem. Here the story takes a different turn:

**Theorem 8.24** *The language $\overline{E}_{TM}$ is recognizable.*

**Proof** There are two kinds of string in $\overline{\text{E}}_{\text{TM}}$. First, there are the strings that are not even a valid encoding of a Turing machine; that is, strings that are not of the form $\langle M\rangle$ where $M$ is a Turing machine. Those strings are easy to detect and accept.

Second, there are the strings that of the form $\langle M\rangle$ where $M$ is a Turing machine and the language of $M$ is not empty. In this case, we will search for a string $w$ that is accepted by $M$. But we need to do this carefully, to avoid getting stuck simulating $M$ on any particular input string.

Here's a Turing machine $S$ for $\overline{\text{E}}_{\text{TM}}$:

1. Test if the input string is of the form $\langle M \rangle$ where $M$ is a Turing machine. If not, accept.

2. Let $t = 1$.

3. Simulate $M$ on the first $t$ input strings for $t$ steps each.

4. If $M$ accepts any string, accept. Otherwise, add 1 to $t$.

5. Repeat Steps 3 and 4 (forever).

Clearly, if $S$ accepts $\langle M \rangle$, then $L(M) \neq \emptyset$. On the other hand, if $L(M) \neq \emptyset$, then $S$ will eventually simulate $M$ on some string in $L(M)$ for a number of steps that is large enough for $M$ to accept the string. Which implies that $S$ accepts $\langle M \rangle$. Therefore, $S$ recognizes $\overline{\mathrm{E}}_{\mathrm{TM}}$. □

Since $\mathrm{E}_{\mathrm{TM}}$ and $\overline{\mathrm{E}}_{\mathrm{TM}}$ are both undecidable, this implies the following:

**Corollary 8.25** *The language $E_{TM}$ is not recognizable.*

## Exercises

8.8.1. Recall that $\mathrm{ACCEPTS}\varepsilon_{\mathrm{TM}}$ is the language of strings of the form $\langle M \rangle$ where $M$ is a Turing machine that accepts the empty string. Show that $\overline{\mathrm{ACCEPTS}\varepsilon_{\mathrm{TM}}}$ is not recognizable.

8.8.2. Show that the class of recognizable languages is closed under union, intersection, concatenation and the star operation.

# Index