## Contents

**Preface**  

1 **Introduction**  

2 **Finite Automata**  
  2.1 Turing Machines  
  2.2 Introduction to Finite Automata  
  2.3 More Examples  
  2.4 Formal Definition  
  2.5 Closure Properties  

**Index**
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 TeX-works 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 defining (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.
2.1. TURING MACHINES

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b, R</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.1: A transition table for a model of computation that’s too simple

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>q₁, b, R</td>
<td>q₀, a, L</td>
<td>q₂, b, R</td>
</tr>
<tr>
<td>q₁</td>
<td>q₁, a, L</td>
<td>q₁, c, R</td>
<td>q₀, b, R</td>
</tr>
<tr>
<td>q₂</td>
<td>q₀, c, R</td>
<td>q₂, b, R</td>
<td>q₁, c, L</td>
</tr>
</tbody>
</table>

Figure 2.2: A transition table for a more realistic model

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 q₀, q₁, q₂ 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-
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
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.
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

<table>
<thead>
<tr>
<th></th>
<th>underscore</th>
<th>letter</th>
<th>digit</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_2$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
</tr>
</tbody>
</table>
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 \texttt{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 \texttt{input-file}, the automaton enters state \( q_1 \) when it sees the first \texttt{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.
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 than 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
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-
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 \( \epsilon \) 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.
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 encounter 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.

![Figure 2.10: A DFA for the language of strings that start with 1](image)

**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
have to always be ready, as if the current symbol was the last one.\footnote{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.”} 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.
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
CHAPTER 2. FINITE AUTOMATA

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 $k$. 

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 $\{\epsilon\}$.

$k$. 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 $\{\epsilon\}$ (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.
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.

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 2.19. 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.
The transition function of the DFA can then be described as follows:

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

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

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

There really was no need to formally describe the DFA of this example because the transition graph of Figure 2.19 is very simple and crystal clear. Here are two additional examples of DFA's. The second one will generalize the first one and illustrate the usefulness of formal descriptions.

**Example 2.15** 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 2.16 Now let’s generalize this. 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 $k = 4$.

It should be pretty clear that a DFA exists for every $k$. But the transition diagram would be difficult to draw and would likely be ambiguous. This is an example where we are better off describing the DFA by giving a formal descrip-
Figure 2.21: A DFA for the language of strings whose digits add to a multiple of 4
tion. 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 \(c \in \Sigma\),

\[
\delta(q_i, c) = q_j, \quad \text{where } j = (i + c) \mod k.
\]

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 2.17** 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

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

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 \textit{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

\(^3\)It is understood here that \(w_1, \ldots, w_n\) are the individual symbols of \(w\).
it means for a DFA to accept a string, that definition is complete. Here it is again, for completeness:

**Definition 2.18** The language recognized by a DFA $M$ (or the language of $M$) is the following set:

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

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 2.19** 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.

**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?
2.4. **FORMAL DEFINITION**

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 \([x_0 y_0 z_0] [x_1 y_1 z_1] \cdots [x_n y_n z_n]\) 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 \textit{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.
2.5. CLOSURE PROPERTIES

This new language is the complement of the first one.\textsuperscript{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 $L$.

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.\textsuperscript{5}

\textsuperscript{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: $\overline{L} = \{w \in \Sigma^* | w \notin L\}$. In other words, $\overline{L} = \Sigma^* - L$.

\textsuperscript{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 $A \cup B = \{w \in \Sigma^* \mid w \in A$ or $w \in B\}$.\(^6\)

---

\(^6\)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 $A \cup B = \{w \in (\Sigma_1 \cup \Sigma_2)^* \mid w \in A$ 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 either contain an even number of 0’s or end in 1. 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

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

and

$$L_2 = \{ w \in \{0, 1\}^* \mid w \text{ ends in 1}\}$$

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.

In this DFA, the 0 transition coming out of $q_0 r_0$ goes to $q_1 r_0$ because in $M_1$, the 0 transition coming out of $q_0$ goes to $q_1$ and in $M_2$, the 0 transition coming out of $r_0$ stays at $r_0$. Similarly, the 1 transition coming out of $q_0 r_0$ goes to $q_0 r_1$ because in $M_1$, the 1 transition coming out of $q_0$ stays at $q_0$ while in $M_2$, the 1 transition coming out of $r_0$ goes to $r_1$. The remaining transitions can be figured out in the same way.

The accepting states of $M$ are those pairs that have $q_0$ as their first state or $r_1$ as their second state. This is correct since $M$ should accept the input whenever either $M_1$ or $M_2$ accepts.

The above idea can be generalized to show that the union of any two regular
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 end in 1, and a DFA for the union of these two languages.
The class of regular languages is closed under union.

**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 DFAs 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

\[
Q = Q_1 \times Q_2 \\
q_0 = (q_1, q_2) \\
F = \{ (r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2 \}
\]

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 $w \in A_1 \cup A_2$, 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, $L(M) = A_1 \cup A_2$.

\(^7\)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 $w$.\[\square\]
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

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

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 $B$.

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 $(r_n, s_n)$.  

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:

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

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 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 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
2.5. Closure Properties

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

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 \( j \geq 2 \).
Index

ε, 17
alphabet, 17
concatenation, 40
DFA, 26
finite automaton, 9
  formal definition, 25
  formal description, 27
language, 17
model of computation, 5
neutral input symbol, 22
regular language, 32
string, 17
  empty, 17
  length, 17
Turing machine, 7