7.2 Regular Expressions and Finite Automata

The mechanisms that have been developed for specifying formal languages are simple, compact, and elegant. They follow one of two basic approaches. The first is language-based: we assign meaning to a few characters beyond those in the language to allow us to write character strings that specify languages. The second is machine-based: we describe a simple class of abstract machines that each can solve the specification and recognition problems for some language. To introduce these topics, we consider first a language-based approach, then two different machine-based approaches, then relationships among the three.

Regular expressions. Since a formal language is a set of strings, we can use basic operations on sets to give an efficient way of specifying formal languages:

Union. A string is in the union of two sets of strings if and only if it is in one or both of them. For example, the union of \{0, 10\} and \{01, 10, 1\} is \{0, 01, 10, 1\}. Since a formal language is a set of strings, we can list the member strings in any order, and since a formal language is a set of distinct strings, we do not include duplicates in the union. For the binary alphabet, we use the notation \( R \cup S \) to denote the union of \( R \) and \( S \); for other alphabets, we use the notation \( R | S \).

Concatentation. When we write one set of strings after another, we specify a more powerful operation: form the set of all strings that can be created by writing a string from the first set followed by a string from the second set. For example, if \( R \) is the set \{0, 01\} and \( S \) is the set \{0, 10, 101\}, then \( RS \) is the set \{00, 010, 0101, 0110, 01101\}. Again, we do not include duplicates in the result. Note that simply writing a bitstring by putting one character after the other is a special case of this operation.

Closure. If \( R \) is a nonempty set of strings, the notation \( R^* \) specifies an infinite number of strings. A string is in \( R^* \) if and only if it comprises 0 or more strings from \( R \), written one after the other. We might write

\[ R^* = \varepsilon \text{ or } R \text{ or } RR \text{ or } RRR \text{ or } RRRR \ldots \]

but the English-language definition is more precise and just as easy to understand.
A regular expression (RE) is a character string that specifies a formal language. We define what regular expressions are (and what they mean) recursively, using the union, concatenation, and closure operations on sets. Specifically, every regular expression is either

- the empty string (ε), specifying the empty set {}  
- a period (.), specifying the set of characters in the alphabet 
- a symbol in the alphabet, specifying the set consisting of that symbol  

or recursively defined as follows (where R and S are REs)

- R | S (or, for the binary alphabet, R or S), specifying the union of the sets specified by R and S  
- RS, specifying the concatenation of the sets specified by R and S  
- R*, specifying the closure of the set specified by R  
- (R), specifying the same set specified by R  

We immediately need to note two features of this definition. First, it implicitly assumes that the language’s alphabet does not include the characters |, *, (, and ). We refer to these characters as metasymbols and will discuss later how to deal with languages that contain them. Third, the definition can be ambiguous: according to the definition, the RE 1 or 01 could specify the set {1, 01} or the set {11, 01}. As with arithmetic expressions, we use parentheses or rely on a defined operator precedence order to resolve such ambiguities. For REs, the defined precedence order is closure, concatenation, union, and by “using parentheses,” we mean that we can enclose R or S or both in parentheses in the recursive definitions if we want to change the defined precedence. Thus, we represent the set {1, 01} by the RE 1 or 01 and the set {11, 01} by the RE (1 or 0)1. Moreover, the RE 0 or 1* is not the same as .* (but (0 or 1)* is the same as .*). These tiny examples are just warmups: we will be considering many examples of REs throughout this section.

With operator precedence and parentheses to resolve ambiguities, the recursive definition not only gives us a way to build up arbitrarily complex REs, but also it precisely defines what they mean. Every regular expression fully specifies a set of strings (a formal language). Throughout this section, you will see many examples of REs, usually accompanied by informal language descriptions and examples of strings in the language (labeled yes) and strings not in the language (labeled no). To familiarize yourself with REs, you should take the time to convince yourself that each RE you encounter
does specify the language as claimed, first by verifying not only that the strings listed as \texttt{yes} are both in language and specified by the RE but also that the strings listed as \texttt{no} are not in the language and not specified by the RE. Your goal is to understand why all strings in the language and only those strings are specified by the RE. Some of the languages that we consider might seem simplistic and arbitrary to you at this point, but rest assured that REs are widely used in practice and they also set the stage for us to consider some profound theoretical questions of critical importance.

For studying theoretical questions, we use the binary alphabet; using more familiar alphabets opens up a wealth of practical applications.

\textit{Elementary example}: You certainly have encountered a simple and familiar application of REs when solving a crossword puzzle or playing a word game. How many times have you asked yourself a question like “What is a 8-letter word whose middle two letters are \texttt{hh}?” The RE that describes the set of all such 8-letter sequences is just \ldots \texttt{hh} \ldots and once you learn about the mechanisms described in this section for finding which of those is in the dictio-
In this chapter, you will address crossword puzzles and word games from an entirely different point of view.

Example from genomics: The human gene sequence has a region that can be described with the RE `gcg(cgg)\*ctg`, where the number of repeats of the `cgg` pattern is highly variable among individuals. Moreover, a certain genetic disease that can cause mental retardation and other symptoms is known to be associated with a high number of repeats. REs are widely used in practice to address important scientific problems of this sort.

An important application area for REs is within computer science itself, where we are always concerned with describing information in a precise and complete manner. For example, when you type your name, address, and other information into a form on the web, the first action of the program that processes the information is to check that what you typed makes sense. If you typed letters where it expects numbers or a dollar sign where it expected a phone number, it will flag the error and ask you to fix it. Later in this chapter, we will see how to use REs to describe things such as dates, credit-card numbers, Java identifiers, and many other familiar low-level abstractions. More
important, REs are a basic component in the process of compiling programs written in high-level languages such as Java into machine language. Even more important, as we have repeatedly emphasized, REs are the first step down the path of addressing fundamental questions about computation.

We will consider applications in more detail after developing a firm understanding of the basic properties of REs, which is easier to do if we restrict attention to the simple case of the binary alphabet, where languages are sets of bitstrings.

Can we specify any formal language with some RE? No. Perhaps you already noticed that some of the languages described in the previous section are missing, and in the next section we will discuss a formal proof that demonstrates this fact by exhibiting a language that cannot be specified with any RE. But the class of languages that can be specified by some RE is sufficiently important that it has a name: A language is regular if and only if it can be specified with a regular expression. Again, regular languages arise in many important applications, and it is important to know whether or not a given language is regular.

You will not be surprised to learn that languages like Prime and Fermat are not regular, but what about DivisibleByThree or Palindrome? One of our goals is to develop understanding on where we can classify languages. While there are plenty of examples of interesting and useful regular languages, there are also many interesting and useful languages that are not regular. Later, we consider the idea of more powerful specification systems than REs to address non-regular languages.

Some examples of regular expressions over the alphabet of decimal digits

<table>
<thead>
<tr>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>.*5</td>
<td>.*0 divisible by 5</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(0</td>
<td>1)*</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>.*0...</td>
<td>fifth-to-last digit is 0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How can we be sure that each RE specifies the same language as does the informal italicized description? In general, we need to prove that fact, as you saw when trying to understand your first REs (see also Exercise 99). The significance of REs is that they can allow us to move away from the informal descriptions because they provide a way to specify languages that is both natural for an important class of applications and rigorous. Regardless of our confidence in the accuracy of the informal specification, each RE is a precise and complete specification of some regular language. Indeed, we can just write down an RE like 

\[(01^* or 101)^*(11^*1 or (0.))^*\]

without necessarily having any idea what language it specifies.

The complete and precise specification afforded by REs lead immediately to various natural and well-specified problems. For instance, given two REs, how can we check whether or not they specify the same language? It is easy to check that

\[0(1 or 01 or 001)\]

and

\[(0 or 00)(1 or 01)\]

specify the same language, but how about

\[((011 or 1000)^*(10^* or 1*0))^* and (01^* or 101)^*(11^*1 or (0.))^*\]

or some arbitrarily complicated expression that runs to (say) thousands of characters? This problem is the equivalence problem for regular languages. Similarly, when we are asking for 8-letter words from the dictionary whose middle two letters are hh, we are asking for the intersection of two languages: strings in both. REs allow us to precisely define problems like these, but solving them is another matter entirely.

An even more fundamental problem is the following: Given a bit-string, how do we know whether or not it is in the language specified by a given RE? For example, is 0100111101001110 in the language specified by 

\[(01^* or 101)^*(11^*1 or (0.))^*\]?

This problem is the recognition problem for regular languages. Again, the significance of REs is that they allow us to precisely define the problem for a large class of languages (regular languages), but they do not solve it. Certainly, we need to address this problem for regular languages before we begin to use REs in practical applications and before we can begin to address more difficult classes of languages that include problems like Prime, Fermat, or Halt.

**Deterministic finite automata.** To address problems such as the equivalence, intersection, or recognition problems for REs, we consider a simple computational model. At first, this model will seem to you to be totally unrelated to REs, but rest assured that it is the first step on a path towards solving these problems.
An abstract machine is a model of computation for the formal language recognition problem. At the most general level, an abstract machine is nothing more than a mathematical function that maps an input bitstring to a single output bit that we interpret as telling us whether the string is in the language recognized by the machine. We normally consider a slightly more detailed model that considers the input bits individually. Specifically, for any given formal language, we imagine a device that interfaces with the outside world with three components: a start switch; a tape drive for input/output with a head that can read or write one bit at a time and a mechanism that moves the tape right or left under the tape head by one bit position; and two indicator lights for output (yes and no). When we load the machine by putting a given bitstring on the tape (positioned at its left end) and then press the start button, we expect the machine to read the input tape and the yes indicator to light if the bitstring is in the machine’s formal language and the no indicator to light if not. We say that the machine accepts the bitstring if the yes indicator lights and rejects the bitstring if the no indicator lights. Thus, an abstract machine specifies a formal language (the set of all bitstrings it accepts) and presents an abstract solution to the recognition problem (put a string on the tape, press the start button and see whether the yes indicator lights).

We are focusing now on the use of abstract machines in the theory of computation, but you should also be aware that this focus is justified because abstract machines are appropriate for modelling practical computations of all kinds. For example, when you use an automated teller machine, your sequence of button pushes corresponds to an input bitstring, the accept indicator corresponds to your money being dispensed, and the reject indicator to your getting an insufficient funds notice. The birds-eye view of a Java program that we considered at the beginning of this book is another example: we can imagine having one abstract machine for each bit of output. The idea is so general as to encompass any kind of computing that we might encounter. We will consider other (more specific) examples throughout this chapter.

How does an abstract machine decide whether to light the accept or reject indicator? We expect that it has to read some or all of the input and to
do some computing to come to a decision. What kind of computing? Each type of abstract machine is characterized by the basic operations that we define for it. Our goal is to try to strip away nonessential details and focus on types of machines that use a set of basic operations that is as simple as possible.

Abstract machines are special-purpose. Each one solves just one problem: the recognition problem for some formal language. In due time, we shall see how the concept helps us understand the problem-solving power of familiar general-purpose computers. Working with machines that are as simple as possible provides us with the opportunity to address questions that are of very general applicibility. For example, if a particular type of abstract machine is so simple as to model all known computers and then we can show that no abstract machine of that type exists that can solve a particular problem, then we have shown that no known computer can solve the problem!

Certainly, it would seem that any machine needs to be able to test the value of an input bit, so we begin by studying a kind of abstract machine that can do little else, which is known as the deterministic finite automaton (DFA).

Each DFA consists of a finite number of states, each of which is designated as either an accept state or a reject state and each of which also specifies a 0-successor state and a 1-successor state. All DFAs start at state 0 with an input bit-string on the tape and the tape head on the leftmost bit in the input string and changes state in discrete steps according to the following rule: read a bit, move the tape head right one bit position, and then change state to the 0-successor of the current state if the bit read was 0 and to the 1-successor of the current state if the bit read was 1. If the DFA is in an accept state when the input is exhausted, the yes indicator lights; if it is in a reject state, the no indicator lights.

**Graphical representation.** Normally we represent DFAs by drawing a graph. States are labeled nodes, transitions are labeled directed arcs. To indicate that a state is an accept state, we add an unlabeled arc leaving it, with no successor.
(We refer to such arcs as an *end-of-input* transitions.) Such graph drawings are not very different from the flowcharts that we considered when we first considered conditional statements in programming. By simply interpreting each node as meaning “read a bit, then follow the arc that has the same label as the bit” and interpreting the presence or absence of the end-of-input transition on the state you wind up in when the input is exhausted as meaning accept or reject, respectively, you can easily use the graphical representation to check whether or not a given DFA accepts a given input string.

To systematically trace the operation of a DFA on a given bitstring, you can simply draw the nodes and arcs that it takes when trying to recognize the bitstring, as follows: The sequence of arc labels is the input bitstring, and the sequence of node labels is the sequence of state transitions the DFA would make in trying to decide whether to accept or reject that bitstring, with the final state dictating whether the bitstring is accepted or rejected.

For example, our sample DFA, given the input 01100, will behave as follows: Starting in state 0, it reads the first 0 and stays in state 0. Then it reads the second bit and moves to state 1 because that bit is 1. Then it moves to state 2 because the third bit is 1 and stays in state 2 because the fourth and fifth bits are 0. The machine is in state 2 when the input is exhausted, which does not have an end-of-input transition, so it lights the *no* light. In other words, the bitstring 01100 is not in the language recognized by the machine.

What bitstrings are in the language recognized by the machine? To answer this question, we need to study the machine to make inferences about the bitstrings it accepts and rejects. You can see immediately that 0s do not affect the outcome, so we only have to worry about the 1s. It rejects 1 and 11 but accepts 11; it rejects 1111 and 11111 but accepts 111111, it is reasonable to leap to the conclusion that this DFA specifies *DivisibleBy3* (the set of all bitstrings whose number of 1s is divisible by 3). It is not difficult to expand this observation into a formal
proof, by induction of the number of bits read, that the machine is in state 0 iff the number of 1 bits read is a multiple of 3, in state 1 iff the number of 1 bits read is a multiple of 3 plus 1, and in state 2 iff the number of 1 bits read is a multiple of 3 plus 2 (see Exercise 99).

Three more examples of DFAs are illustrated at the top of this page and several others are described in the exercises. Again, these examples are worthy of your careful study. For each one, convince yourself that it recognizes the claimed language by first tracing its operation to check that it accepts a few strings that you know are in the language and rejects a few strings that you know are not in the language, then trying to understand why it always does so.

From these few examples, you should be persuaded while DFAs are very simple to define, understanding their behavior is hardly a simple task. Indeed, the DFA abstraction is widely used to address nontrivial computational problems in numerous practical applications: in text editors for pattern
matching, in compilers for lexical analysis, in web browsers for .html parsing, in operating systems for graphical user interfaces, and in many other software applications. The DFA abstraction also suffices to describe control units in physical systems such as vending machines, elevators, automatic traffic signals, and computer processors.

An example of a control unit: Consider a vending machine that accepts nickels, dimes and quarters, for items that cost twenty-five cents. The behavior of the control unit for such a machine is easily understood with a DFA model. If you suppose that the machine has an internal state corresponding to each possible amount of money inserted, then it easy easy to see how the machine must change state each time a coin is inserted to reflect the value of the coin added. If we label the states such that we can interpret state $i$ to mean that amount of money so far inserted is five times $i$, then the necessary state transitions are straightforward (inserting a dime takes the machine from state $i$ to state $i+2$, and so forth), and the machine accounts for all possible ways to insert coins that total twenty-five cents. For example, the state transition sequence 0-2-3-5 corresponds to inserting a dime, then a nickel, then a dime.

When we say that we are speaking of an abstract machine, we mean that we do not specify how the machine is realized in the physical world. This decision gives us the freedom to work with various different DFA representations in various different circumstances. We can fully specify a DFA with a table of numbers, or as a Java program, or as a circuit, or as a piece of mechanical hardware. Indeed, DFAs are appropriate for modeling all kinds of objects found in nature. The DFA abstraction allows us to reason about specific properties of all of these diverse mecha-
nisms, both real and abstract, at the same time. Considering various different representations will help you to better understand the nature of the DFA abstraction.

**Tabular representation.** Any DFA can be fully specified as a table with three pieces of information for each state: its accept/reject bit, its 0-successor, and its 1-successor. Normally, we simply use the row number to index the table and to name the states. This representation is a way to fully describe any particular DFA, but it does not address the issue of specifying how DFAs work. To handle languages with bigger alphabets, we need a column in the table for each possible character in the alphabet.

<table>
<thead>
<tr>
<th>name</th>
<th>accept?</th>
<th>0-successor</th>
<th>1-successor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>false</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>false</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

**Tabular representation of DivisibleBy3 DFA**

**Bitstring representation.** Since the tabular representation is just $N$ rows that each consist of a bit and two numbers less than $N$, we can convert the values to binary and then collapse them all together to represent the whole DFA as a bitstring. The only difficulty in this process is specifying how many bits to use to encode the numbers. One trick for accomplishing this is to include, as the first bits in the string, an encoding of the number of bits to be used for state names in unary: we put a 1 for each bit needed to encode $N$ in binary, followed by a single 0 (to mark the end of the unary number). For example, if $N$ is 27, then we would start the bitstring with 111110 because 27 is 11011 in binary, so we need five bits to encode $N$ (and all the state names, which are less than $N$). Here is the code for our DivisibleBy3 DFA in this scheme:

```
1 1 0 1 1 0 0 0 1 0 0 1 1 0 1 1 0 0 0
```

This coding is yet another example of our ability to use bitstrings to model any discrete computational problem whatsoever. Indeed, we can phrase basic questions that we might have about DFAs as language questions: we could speak of the language of all bitstrings that represent 5-state DFAs, or the lan-
The tabular representation makes clear that every possibility must be accounted for in a DFA, which is one reason they are widely used in building control units and other devices that have to handle all possibilities (including some that a designer might not otherwise consider). For example, you may have noticed that the vending-machine DFA example that we just considered did not account for the possibility that someone might insert pennies, or more than twenty-five cents. Using the tabular representation makes it easy to design a DFA for a vending-machine control unit that accepts pennies (but ignores their value) and coins whose value totals twenty-five cents or more.

**Improved vending-machine DFA**

<table>
<thead>
<tr>
<th>name</th>
<th>accept?</th>
<th>penny</th>
<th>nickel</th>
<th>dime</th>
<th>quarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>false</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>false</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>false</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>true</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Mathematical representation.** With mathematical notation, we can expand the DFA concept into a complete formal model of computation (see the Q&A at the end of this section for details). The advantage of the mathematical language is that mathematicians are accustomed to making complete and precise statements with such notation, and it is the language of choice for detailed proofs. Normally, we will use representations that are more informal and intuitive than this one, but you should be aware that we could use such notation to validate our statements with formal mathematical proofs, if desired. Actually, we often prove things by providing Java code, which is better than a mathematical proof in some ways, because you can run it to check its validity.

**Java code representation.** A programming language such as Java is also a formal notation, and it is not unreasonable to use Java code to specify DFAs. There are many ways to do so: we begin with a simple representation that not just specifies a DFA, but also is the code to simulate its operation. We use a class `Tape` to represent the input tape and invoke the method `isEmpty` to check whether the input is exhausted or `read` to get another input bit. Using
this abstraction, we can encode the tabular representation of any DFA in Java
code, in the following manner:

```java
Tape tape = new Tape();
int pc = 0; while (!tape.isEmpty())
{
    boolean bit = (tape.read() == 0);
    if (pc == 0) {
        if (bit) pc = 0;
        else pc = 1;
    }
    else if (pc == 1) {
        if (bit) pc = 1;
        else pc = 2;
    }
    else if (pc == 2) {
        if (bit) pc = 2;
        else pc = 0;
    }
}
if (pc == 0) System.out.println("false");
if (pc == 1) System.out.println("false");
if (pc == 2) System.out.println("true");
```

You can easily identify the tabular representation of our `DivisibleBy3` DFA in
this code, and it also serves as a simulator of that DFA. For example, it will
print `false` if the successive calls to `tape.read()` returns the sequence `1 0
1 1 0 1 0 0` because the number of 1s in that bitstring is not divisible by 3.
Our use of the name `pc` for the current state is deliberate: that variable
functions just like the program counter in a machine-language program. Clearly,
we can write a program like this to simulate any DFA, and such a program is
quite useful to have for studying DFAs. For example, it immediately yields an
automatic way to produce traces (add a statement to the loop that prints out
the value of `pc`), so that we do not have to do them by hand.

---

**Machine-language code representation.** It is easy to imagine a machine like TOY
that is built to solve any problem that any DFA can solve. Such a computer needs
only one instruction, with three fields: an accept/reject bit, a 0-successor address,
and a 1-successor address (no opcode is necessary because the machine has only
one instruction!). The semantics of the instruction directly mirrors the DFA definition: if the input is exhausted, then
light the accept indicator or the reject indicator according to the accept/reject bit in the current instruction; otherwise read an input bit and branch to the
0-successor address if the bit is 0 and to the 1-successor address if the bit is 1.
A DFA is nothing other than a program for this simple machine, and the task of creating a DFA is therefore nothing other than (machine-language) programming. Creating such a machine from gates and flip-flops would certainly be much easier than doing so for TOY. When we talk about the computational power of the DFA model, we are talking about the power of such a machine.

Is there something that we can do with some DFA that we cannot do with a conventional computer or a Java program? Practically speaking, the Java-code representation that we just considered demonstrates that the answer to this question is no. But what is the set of languages that can be recognized by a DFA? Can we have a DFA that recognizes \textit{Prime}, \textit{Fermat}, or \textit{Halt}? What do we do when presented with a DFA so huge that it will not fit into the particular computer that we are using? It seems reasonable to assume that the input tape has unbounded capacity (that we can always get more), but having to build a bigger machine to solve a bigger problem seems burdensome. Our goal in considering formal models of computation is to be able to address such questions and to relate them to similar practical questions that we might face. As a warmup, we consider the idea of taking a DFA representation as input to a Java program.

\textit{DFA simulator}. Specifically, it is easy to write a Java program that simulates any DFA. Program 7.2.1 takes a DFA and bitstring as input (a program for the machine and the input to that program) and prints \texttt{true} or \texttt{false}. This program represents a complete specification of what constitutes a DFA, subject to the assumption that we can use arrays whose size is the number of states in the DFA. There is nothing special about the Java implementation; we could do the same in TOY without much more code. Regardless of any theoretical implications, Program 7.2.1 is an indispensible tool for the study of properties of particular DFAs: you provide a DFA’s tabular description and an input bitstring and Program 7.2.1 will tell you whether or not the DFA accepts the bitstring. It is a practical solution to the recognition problem for DFAs.

Writing a Java program that takes a DFA as input is a conceptual leap that opens up the consideration of numerous intriguing questions that we might wish to write programs to solve. Are two given DFAs equivalent? Can we compute a DFA that recognizes the same language as a given DFA but has fewer states (or report that none exists)? What are the chances that a random...
Program 7.2.1  DFA recognition

```java
public class DFA {
    private static final int START = 0;
    private int N;
    private boolean[] accept;
    private int[][] next;
    DFA(In in) {
        N = in.readInt();
        accept = new boolean[N];
        next = new int[N][2];
        for (int i = 0; i < N; i++) {
            accept[i] = (in.readInt() == 1);
            next[i][0] = in.readInt();
            next[i][1] = in.readInt();
        }
    }
    public boolean simulate(Tape tape) {
        int pc = START;
        while (!tape.isEmpty())
            pc = next[pc][tape.read()];
        return accept[pc];
    }
    public static void main(String[] args) {
        DFA dfa = new DFA(new In(args[0]));
        Tape tape = new Tape(args[1]);
        System.out.println(dfa.simulate(tape));
    }
}
```

The constructor for this DFA class takes as input a DFA represented as an integer $N$ followed by $N$ (accept/reject, 0-successor, 1-successor) triples and builds a state-indexed-array representation of the DFA. The simulate method decides whether or not the string on the tape is in the language accepted by the DFA.

% more mod3.txt
3 1 0 1 0 1 2 0 2 0
% javac DFA
% java DFA mod3.txt 10010011
false
% java DFA mod3.txt 100110001
true
% more lt3
4 1 0 1 1 2 1 2 3 0 3 3
% java DFA lt3 10001000
true
bitstring will be recognized by a given DFA? With the bitstring DFA representation, we could frame all such questions as language questions, and also consider the idea of DFAs that operate on DFAs. Is there a DFA that recognizes whether a given

As an example, consider the problem of writing a Java program to solve the following natural problem: Does a given DFA recognize any bitstrings at all? This question is equivalent to asking if there is any path at all from the start state to some accept state in the DFA. As it turns out, we have already addressed this question at a more abstract level, when we studied the graph abstraction and graph-search algorithms: it is equivalent to considering the DFA to be a graph and looking for a path in the graph. Accordingly, we might add the following method to the DFA class of Program 7.2.1:

```java
public boolean isEmpty()
{
    Queue Q = new Queue();
    boolean[] reachable = new boolean[N];
    Q.put(START);
    while (!Q.empty())
    {
        int t = Q.get();
        if (accept[t]) return false;
        reachable[t] = true;
        if (!reachable[next[t][0]]) Q.put(next[t][0]);
        if (!reachable[next[t][1]]) Q.put(next[t][1]);
    }
    return true;
}
```

This code adapts breadth-first-search, but we could use any other graph-search method, as well.

One purpose in considering this example is to introduce you to the idea that graphs are appropriate for modeling DFAs and therefore graph algorithms are useful when writing programs to process DFAs. We will apply this same idea to solve a more difficult problem in the next section.

**Nondeterminism.** In this section, we consider an extension to the DFA model that does not correspond to the way that we normally think about
computers in the real world. Why do we do so? This model not only leads to more compact automata that are more convenient to work with, but also it provides insights that both help us build useful practical programs and serve as the basis for addressing profound theoretical questions. In particular, it represents a link between DFAs and REs that leads us to a full understanding of regular languages and to a practical solution to the recognition problem for REs.

The behavior of a DFA is *deterministic*: for each input bit and each state, there is exactly one possible state transition. A *nondeterministic finite automaton* (NFA) is an abstract machine that allows other possibilities.

Each NFA consists of a finite number of states, each of which specifies any number of null-successor states, any number of 0-successor states and any number of 1-successor states. All NFAs start at state 0 with an input bitstring on the tape and the tape head on the leftmost bit in the input string and changes state in discrete steps according to the following rule: either change state to any null-successor state of the current state or read a bit, move the tape head right one bit position, and then change state to any 0-successor of the current state if the bit read was 0 and to any 1-successor of the current state if the bit read was 1. If the NFA is in state 1 when the input is exhausted, its *true* indicator lights; otherwise, its reject indicator lights.

Not only can an NFA change state without reading an input symbol, but also, given a current state and input symbol, an NFA can have zero, one, or several possible transitions from each state corresponding to each input symbol. The mechanism for choosing whether or when to make a transition is unspecified. We regard an input as acceptable if there is any sequence of transitions that can take the machine from the start state (0) to the accept state (1). Unlike DFAs, we allow only one accept state.

For example, the NFA pictured here is nondeterministic for two reasons: when in state 0, the next state might be either state 0 or state 2 after reading in a 0; when the automaton enters state 1, there must be no more input symbols to read since there are no transitions from state 1.
Without the simple state transition rules that characterize DFAs, we have to do more work to solve the recognition problem for NFAs. For example, suppose that the input string is 10001: if you study it for a moment, you will see that this NFA can recognize this input via the state transitions 0-0-0-2-1. It could also make the transitions 0-2-1 or 0-0-0-0 but those sequences do not both exhaust the input and lead to state 1 and are therefore not of interest. On the other hand, it is easy to see that if the input string is 1111, there is no way for the automaton to get from state 0 to state 1, so it must light the false indicator. To solve the recognition problem, we have to write a program that checks all the possibilities. The necessity to do so contrasts sharply with the corresponding task for DFAs, where we follow simple rules to go from state to state.

This NFA recognizes the language of all strings whose second to last symbol is 0. It can accept any string whose second to last symbol is 0 by choosing to stay in state 0 (the start state) until the second to last character is read, at which point it reads the 0 and advances to state 2 and then state 1 (the accept state) after reading in the last symbol. There is no sequence of state transitions that could accept any string whose second to last character is not a 0. It is easy to see how to generalize this NFA to a \((k+1)\)-state NFA that recognizes the language of all strings whose \(k\)th to last bit is 0. By contrast, it is known that any DFA that recognizes this language must have at least \(2^{k-1}\) states.

NFAs can also have null transitions, which allow them to change state without even reading an input bit. An example of such an NFA is pictured at the bottom of this page. This capability would seem to make it even more difficult for us to understand whether or not a particular NFA accepts a particular string, let alone what language a particular NFA recognizes. To appreciate this claim, take the time to convince yourself that the pictured NFA accepts only bitstrings that do not contain the substring 110, as claimed.

As with DFAs, we do not specify how the automaton does its computation, and, indeed, it is not easy to imagine how we would build a machine like this. It would seem to be quite a bit of work for each machine not only to find a sequence of transitions from start state to accept state for inputs that are in its language, but also to prove to itself that there is no such sequence of transitions for inputs not in its language.
One way to think about the operation of an NFA is as guessing a sequence of transitions to the accept state: it finds the sequence if there is one. Our intuition is that machines cannot guess the result that we want them to compute, but nondeterminism amounts to imagining that they can do so. Some of the most intriguing questions of theoretical computer science have to do with the power of nondeterminism: if we could build nondeterministic machines, would they be more powerful than the deterministic machines that we can build? We will address this question for finite automata after examining NFAs in more detail.

Representations. To draw an NFA, we can draw a graph with labeled nodes for states and labeled arcs for transitions, just as we did with DFAs, but multiple arcs with the same label can exit a given state, and arcs may have null labels. Such a drawing is quite different from a flowchart! A tabular representation has to include a column for null transitions, and to allow for entries having multiple states. A simpler possibility is to use an arc-list representation, where we just list the arcs (with their labels). We might also imagine a machine-code representation (see Exercise 99), but not necessarily a TOY-like architecture for the machine, because its basic operation has to involve branching to one of two different addresses, with the machine having to guess this correct one. In a Java-like language we would need a construct like

```
    do either { this statement } or do { that statement };
```

These representations make it seem as though nondeterminism is taking us away from familiar computational artifacts. Actually nondeterminism is a
concept that is both of practical utility and also critically important in understanding the fundamental nature of computation.

**Mathematical function representation.** As with the English-language description, the mathematical description of NFAs is only slightly different than of DFAs. Whether or not you want to check the details, you can see that the mathematical description of NFAs is as simple or simpler than the mathematical description of DFAs, and makes the concept of non-termination somewhat more concrete than thinking about machines that guess. The key distinction is that the wording “there exists” covers a much bigger set of possibilities for NFAs than for DFAs.

How can we solve the recognition problem for NFAs (systematically check whether or not a given NFA accepts a given input)? For DFAs, the process was trivial because there is only one possible state transition at each step, but for NFAs, we are looking for the existence of at least one sequence of state transitions in the face of a large number of possibilities. Fortunately, it is not difficult to keep track of all possible state transitions, as follows:

- Start by finding all possible states that can be reached by null transitions from the start state. This is the set of possible states that the NFA could reach before reading the first character.

- Find all states that can be reached by an arc labeled with the first input character, then all states that can be reached from one of those by null transitions. This is the set of possible states that the NFA could reach before reading the second character.

\[
\begin{align*}
\text{all states reachable} \\
\text{after reading i symbols} \\
\text{possible transitions on} \\
\text{reading (i+1)st symbol c} \\
\text{possible null transitions} \\
\text{before reading next symbol} \\
\text{all states reachable} \\
\text{after reading i+1 symbols}
\end{align*}
\]

*One step in simulating an NFA*
• Iterate this process, keeping track of all possible states that the NFA could reach before reading each input character, until the input is exhausted.
• If state 1 is in the set of states left after the input is exhausted, then the input bitstring is in the language accepted by the NFA.
• If state 1 is not in the set of states left after the input is exhausted, or if the set of possible states becomes empty before the input is exhausted, then the input bitstring is not in the language accepted by the NFA.

**NFA trace example.** Consider the operation of our NFA for the set of all bitstrings not containing 110, on the input 01011. We start with the set of states \{0, 1\} since 0 is the start and there is a null transition to 1. The set of possible states after reading the first bit of input (0) is still \{0, 1\} since the only 0-successor arc is from 0 back to 0 and then a null transition to 1 is possible. After reading the second bit of input (1), we could be in state 2 (a 1-successor of 0) or in state 1 (a 1-successor of 1). The third bit is 0, and the only 0-successor from \{1, 2\} is 0, and a null transition is always possible from 0 to 1, so we wind up with \{0, 1\} as the only possible states after reading the third bit. As before, 1-successors from \{0, 1\} take us to \{1, 2\} after the fourth bit, and then the 1-successor from state 1 takes us to \{1\} after the fifth bit. Since it can exhaust the input and end in state 1, we know that this NFA will accept this input (and we can exhibit the sequence of state transitions that the machine would make to do so: 0-0-2-01-1-1). If the input were to have one more bit that is 0, the set of possible states after reading it would be
empty (since the only possible state and it has no \(0\)-successors) so we would know that \(010110\) would be rejected.

**NFA simulator.** For DFAs, we developed in Program 7.2.1 a solution to the recognition problem by simulating the operation of the DFA, step by step. How can we do the same for NFAs? The method of tracing that we just considered gives the basic idea: rather than keeping track of a single state as the program counter, we keep track of a set of states: the set of all states the NFA could reach after having read the tape up to the current character.

To implement this basic idea, we first have to make a number of decisions. First, how should we represent the NFA for input to the program? Perhaps the simplest is to use the arc-list representation: a pair of integers \(N\) (number of states) and \(M\) (number of arcs) followed by a sequence of \(M\) triples, each triple specifying an NFA arc with three integers: its label, source, and destination. For simplicity, we use integer labels, with 2 representing the label for null transitions. Second, what internal representation should we use? Using a graph representation is entirely appropriate, as we saw when considering the problem of determining whether a DFA accepts any string, so we will use our standard Graph ADT (see Chapter 99). For NFAs, the program code turns out to be substantially simpler if we use an array of three graphs, one for each possible edge label (0, 1, or null). When we are considering a specific input character, this representation immediately yields the relevant arcs and spares us from having to search for them. Building this representation from the sequence-of-arcs input format is simple: for each arc, use the label to index the array to choose the graph, then call the `addEdge` method for that graph, with the source and destination as arguments. Program 7.2.2 gives the details of an NFA constructor implemented along these lines.

With these decisions implemented, we can simulate the operation of an NFA with a simple loop based on keeping track of the set of states the machine could have reached reading the input so far in a `UniQueue` data structure, as shown in Program 7.2.3. Recall that the `adjQ/reachable` methods from our standard graph ADT return a `Queue` that contains all nodes adjacent to/reachable from the given vertex, respectively. Thus, we can initialize `pcQ` with the set of all nodes that can be reached via null transitions from `START` by invoking `reachable` for the null-transition graph, with argument `START`. Then, we can update `pcQ` in each iteration of the loop (after reading an input character \(ch\)) in two steps: First, we need to collect all the states reach-
Program 7.2.2 NFA recognition.

```java
public class NFA
{
    private static final int START = 0;
    private static final int ACCEPT = 1;
    private static final int NULL = 2; // alphabet size
    private int N, M;
    private Graph next[];

    NFA(In in)
    {
        N = in.readInt(); M = in.readInt();
        next = new Graph[NULL+1];
        for (int i = 0; i <= NULL; i++)
            next[i] = new Graph(N);
        for (int i = 0; i < M; i++)
        { int label = in.readInt();
            int v = in.readInt();
            int w = in.readInt();
            next[label].addEdge(v, w);
        }
    }

    public boolean simulate(Tape tape)
    // See Program 7.2.3
    public static void main(String[] args)
    {
        NFA nfa = new NFA(new In(args[0]));
        Tape tape = new Tape(args[1]);
        System.out.println(nfa.simulate(tape));
    }
}
```

The constructor for this NFA class builds an array of graphs, one for each alphabet character and one for null transitions, from a sequence of arcs that are represented as (label, from, to) triples. The simulate method decides whether or not the string on the tape is in the language accepted by the NFA (see Program 7.2.3).
able via a ch-transition from some state in pcQ. This is easily accomplished by using adjQ to add the adjacency list of each state on pcQ to a temporary Unique-queue (nextQ). Second, we need to collect all the states reachable via a null transition from some state in pcQ. This is easily accomplished with reachable, as at the beginning. Notwithstanding the significance of the computation that it performs, this program is worth examining carefully, because it represents a small amount of code for a sophisticated calculation. It illustrates the power of abstraction, because it is made simple by virtue of the fact that it makes use of basic abstract data types that we have studied.

The idea of machines that can guess the right answer is a fantasy, but simulation allows us to study and use nondeterminism as if it were real. Remarkably, it is an very usable abstraction in practice, and we will use it to solve the recognition problem for REs. Next, we examine the relationship between automata and regular expressions that enables this use.

**Kleene’s theorem.** There is a striking connection between regular expressions, DFAs and NFAs that has dramatic practical and theoretical consequences. This connection is the first major theoretical result that we consider. Whether or not you are comfortable with the idea of mathematical proof, you should pay close attention to how we establish this connection. We will give the intuition behind each proof, along with enough details that people who are comfortable with proofs can fill in the rest.

Our first proofs are constructive, and perhaps easier to check than some other kinds of proofs (which, for example, might be dependent on principles of logic) because we include examples to illustrate the constructions. While examples does not constitute a proof any more than test cases demonstrate that a program is valid, that are a necessary first step. If you read a proof, then study an example, then reread the proof, you will find it easier to convince yourself of the truth of the claimed statement. Another good approach is to imagine yourself convincing someone else that it is true.

**NFAs are equivalent to DFAs.** This statement is the same as saying that given any automaton of one type there exists some automaton of the other type that recognizes the same language.

Given a DFA, how do we convert it to an NFA? The states and transitions are no problem: a DFA is like an NFA with no null transitions and one transition of each type leaving each state. The only difference is in our conventions that a DFA can have multiple accept states, while an NFA has just
AN INTRODUCTION TO COMPUTER SCIENCE

one accept state, state 1. To convert a DFA to an NFA that recognizes the same language, therefore, we just renumber state 1 to N, add a new state 1, and add null transitions from each accept state to state 1.

Given an NFA, how do we convert it to a DFA that recognizes the same language? We build a DFA that has one state corresponding to each possible set of states in the NFA. There is a uniquely defined set of possible states for any given NFA after having read any given portion of its input, so this DFA

Program 7.2.3 NFA simulation.

```java
public boolean simulate(Tape tape) {
    UniQueue nextQ, pcQ = new UniQueue(N);
    pcQ.put(next[NULL].reachable(START));
    while (!tape.isEmpty())
        { int ch = tape.read();
            nextQ = new UniQueue(N);
            while (!pcQ.isEmpty())
                nextQ.put(next[ch].adjQ(pcQ.get()));
            pcQ = new UniQueue(N);
            while (!nextQ.isEmpty())
                pcQ.put(next[NULL].reachable(nextQ.get()));
            }
    while (!pcQ.isEmpty())
        if (pcQ.get() == ACCEPT) return true;
    return false;
}
```

This implementation keeps track of possible NFA states before reading each character in pcQ. First, pcQ is initialized with the states reachable by null transitions from START. Then, each iteration of the loop puts all states that can be reached by a ch-transition from any state in pcQ onto nextQ, then puts all states that can be reached by a null transition from any state in nextQ back onto pcQ, ready for the next iteration.

% more NFA0x.txt
3 5
0 0 0 0 0 2 1 0 0
0 1 2 1 1 2
% javac NFA
% java NFA NFA0x.txt 100110011
false
% java NFA NFA0x.txt 10010001
true

AN INTRODUCTION TO COMPUTER SCIENCE
can handle all the possibilities that can arise. To figure out the transitions, we use the same approach as in the inner loop in Program 7.2.3. Given a set of NFA states and an input character, we form a set of NFA states by taking the union of all NFA states reachable from any NFA state in the original set by one arc labelled with the character and any number of null arcs. This set of NFA states corresponds to a single DFA state, and is the destination. For every state in the DFA, we can compute a unique 0-successor and a 1-successor in this way. Finally, we define to be an accept state every DFA state whose corresponding NFA set of states contains state 1. If you work carefully through just one example, you will see how this construction works.

There is no formal language that can be recognized by some DFA and not by any NFA and there is no formal language that can be recognized by some NFA and not by any DFA. A priori, you might have thought that non-deterministic finite automata (which can guess the path to the computation) would be more powerful than deterministic ones, but that is not the case. They are equivalent with respect to computational power.

There is one caveat in this story: the DFA corresponding to an N-state NFA might have \(2^N\) states, since that is the number of subsets of an N-element set. Dealing with the cost of computation is another story, which we will address in due time. For now, we are focusing now on what is possible with various sorts of machines.

<table>
<thead>
<tr>
<th>name</th>
<th>accept?</th>
<th>0-successor</th>
<th>1-successor</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td></td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>false</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>true</td>
<td>012</td>
<td>02</td>
</tr>
<tr>
<td>012</td>
<td>true</td>
<td>012</td>
<td>02</td>
</tr>
</tbody>
</table>

Converting an NFA to a DFA
REs are equivalent to NFAs. It is another surprising fact that any language that can be described by an RE can be recognized by some NFA, and any language that can be recognized by an NFA can be described by some RE. Since NFAs and DFAs are equivalent, we can make these same statements about DFAs. Again, we can convince you of this equivalence by describing explicit constructions. We use NFAs for this purpose because they are more flexible. Indeed, this use is an important illustration of the utility of nondeterminism.

To construct an RE that describes the language recognized by any given NFA, we proceed as follows: First, we extend the NFA model to allow arc labels to be REs. For example, suppose that we have two arrows, one labeled 0 and one labeled 1, whose source and destination are both the same. Clearly, we can replace them by a single arc labeled "0 or 1". Similarly, having an arrow labeled 0 from a state to itself is the same as labeling the arrow 0*. With a more systematic treatment of such equivalencies, we can transform any extended NFA to an RE, by removing states. Specifically, given an extended NFA, we can remove any state x in four steps:

- Combine arcs that enter x from the same source using or
- Combine arcs that exit x to the same destination using or
- The previous operations leave at most one self-loop (refer to its label as R). For each entering/exiting arc pair (refer to their labels as S and D respectively) create an arc labeled S R* D that goes from the source of the entering arc to the destination of the exiting arc (and thus skips the state).
- delete the state and all arcs that enter and leave it.

It is easy to verify that these operations yield an extended NFA that is equivalent to the original. The new NFA may have more arcs that the original, but it has one fewer state. Therefore, we can apply it until there are no states left (just a single arc). The label on the arc is an RE that describes the language recognized by the NFA.

No matter what NFA you start with, the above paragraph tells you how to remove a state. But then it tells you how to remove another state, and
so forth until there are no states left, and you are left with an RE. This process is sufficiently well-specified that we could write a Java program to do the conversion, but our interest here is just in the proof: you can convert any NFA into an RE that describes the language recognized by the NFA.

Similarly, you can find an RE that describes the language recognized by any DFA, by first converting the DFA to an NFA (renumber state 1 to N and add null transitions from all the accept states to state 1) and then following the same procedure. You are encouraged to try an example or two.

The second part of the proof that REs and NFAs are equivalent is to show how to construct an NFA that recognizes the same language as any given RE. The same basic procedure, but working backwards, accomplishes this goal. We work from a trivial extended NFA with the RE on a single arc to build an NFA by adding states to simplify the REs on the arcs until each arc has no label, 0, or 1. We start with a single-state extended NFA with an entering arc and an exiting arc labeled with the given RE. Now, every arc with a nontrivial label must be of the form (R), RS, R+S, or R* (where R and S are REs) and we can therefore simplify in one of the following ways.

- If an arc label is of the form (R), remove the parentheses.
- If an arc label is of the form RS, construct a new state and replace the arc by a new state and two arcs, one labeled R from the source to the new state and the other labeled S from the new state to the destination.
- If an arc label is of the form R+S, replace the arc by parallel arcs, one labeled R and the other labeled S.
- If an arc label is of the form of the form R*, replace the arc by an unla- beled arc and a self-loop labeled R.

Each of these constructions leaves an extended NFA that recognizes the same language as the original and simplifies some arc, so we can continue to apply
one of them until each arc label is blank, 0, or 1, leaving us with an NFA. Again, it is up to you to convince yourself that you could use this method to build an NFA corresponding to any given RE. The practical significance of this construction is that it leads to a way to solve the recognition problem for REs: build an NFA, then use Program 7.2.2 to simulate it.

The theoretical significance of the three constructions just described is that they prove REs, DFAs, and NFAs all to be equivalent. Given an instance of any one of the three, we can construct an instance of either of the others that recognizes the same language. (Quick question: How could we construct a DFA corresponding to a given RE? Quick answer: Build an NFA corresponding to the RE using the method just described, then build a DFA corresponding to the NFA using the subset construction.) This equivalence is known as Kleene’s theorem, named after the logician who established the result in 19??.

We have described Kleene’s theorem in detail to get you used to the idea of proving facts about abstract devices, to illustrate that even the simplest machines can lead to surprising theoretical results (nondeterminism does not add power to finite automata!), to introduce you to an interesting and useful computational model that is well-understood, and to introduce you to a basic approach that carries through to support the truly profound ideas at the foundation of the theory of computing.

Converting an NFA to an RE

An Introduction to Computer Science
Kleene’s theorem is more than a significant theoretical result; it is directly applicable in practice. In the next section, we will consider ways in which the theory that we have described so far manifests itself in Java programming. Then, we will return to the theory by considering the idea of limitations on the power of abstract machines (and language models).

**Practical applications.** Even though these computational models are all abstract, they are the basis for widely-used practical tools. REs are nothing more nor less than a simple programming language. You certainly are familiar with the concept that choosing the proper tool is important for any problem-solving activity. After you learn how to use them, you will want to use simple languages like REs (not general-purpose languages like Java) for certain computational tasks.

**RE recognition.** The fundamental practical application of the theory is known as generalized regular expression pattern matching (*grep* for short). Since its introduction by Thompson in the UNIX operating system in the 1970s, it has endured as the quintessential programmer’s tool, and people use it for all sorts of other applications. In its basic form, it is nothing other than a solution to the recognition problem for REs: given an RE and a string, is the string in the language specified by the RE? While details are not always fully specified, practitioners typically use the term *pattern matching* as meaning *RE recognition*.

Our proof of Kleene’s theorem provides a direct path to building a program that solves the recognition problem for REs.
Program 7.2.4  RE recognition.

```java
public class NFA
{
    private static final int START = 0;
    private static final int ACCEPT = 1;
    private int N;
    private String alphabet;
    private int NULL; // alphabet size
    private Graph next[];
    private void NFAbuild(int from, int to, String re)
        // See Program 7.5
    NFA(String ab, String re)
    {
        alphabet = ab;
        NULL = alphabet.length();
        next = new Graph[NULL+1];
        for (int i = 0; i <= NULL; i++)
            next[i] = new Graph(1+re.length());
        N = 2;
        NFAbuild(0, 1, re);
    }
    public boolean simulate(Tape tape)
        // See Program 7.2
    public static void main(String[] args)
    {
        NFA nfa = new NFA(args[0], args[1]);
        Tape tape = new Tape(args[2]);
        System.out.println(nfa.simulate(tape));
    }
}
```

This class provides a solution to the recognition problem for REs over general alphabets. It builds an array-of-graphs representation for the NFA and uses the same simulate code as before. The driver takes an alphabet, an RE, and a string from the command line and prints whether the string is in the language described by the RE.

% javac NFA.java
% java NFA 01 ".*0..." 010111
true
% java NFA actg cgg atacgacgga
true
% java NFA abehilmnoprstuvyz
".*hh.*" triphenylphosphine
false
• Build an NFA that corresponds to the given RE.
• Simulate the operation of the NFA on the given input bitstring.

We have already implemented the second part of this process—that is Program 7.2.3—so we just need to write a program for the first part. One easy way to package this program is as a constructor for the NFA class of Program 7.2.2, as shown in Program 7.2.3. For simplicity, this code is designed for REs with no parentheses; after considering how it works, we will discuss extending it to handle parentheses.

The constructor implementation in Program 7.2.3 is based on the same transformations that we introduced when proving Kleene’s theorem. For that proof we demonstrated that we could build an NFA corresponding to any RE by choosing from among a number of transformations that simplified the RE on some edge in an extended NFA; for this program we need to pick specific transformations and apply them.

The program is based on a recursive program that takes as argument an edge in a generalized NFA. The edge is labeled with a string which must be one of the following:

• a single character or .
• of the form $R|S$ where $R$ and $S$ are REs
• of the form $c^*S$ where $a$ is a single character or . and $S$ is an RE
• of the form $cS$ where $a$ is a single character or . and $S$ is an RE

In the first case, we simple add the specified edge to the NFA (an edge for every character in the alphabet if the character is .); in each of the others, we also know precisely how to proceed, recursively. For the last case, we create a new NFA state and replace our edge by two edges: one labeled $c$ from our source to the new state, and one labeled $S$ from the new state to our destination. These recursive calls correspond to the concentration transformation that we considered when proving Kleene’s theorem. The transformations that we need for the or and closure operations are similar. For the proof we were interested in knowing that at least one of these transformations always applies. This same observation is critical to knowing that the program does not fall into a recursive loop. In general, there may be several transformations to try: in our (deterministic) program, we make the specific choice of checking the transformations in the order listed (using the leftmost or operator).

You could write a program that makes a different choices, but it is important
to be convinced that all cases are covered and that there always is a way to reduce to a smaller RE.

Extending the implementation to handle parentheses involves checking for the following cases:

- the RE is of the form (R)
- there is a | that is not enclosed in parentheses
- the RE is of the form (R)*S
- the RE is of the form (R)S

The first case is trivial (make a recursive call with the parentheses stripped off the string argument); the second case involves just ignoring the | (so as to avoid, for example, splitting an RE such as (0 | 1)* into (0 and 1)*); and the latter two cases involves extending our code for closure and concatenation to use the string R instead of a single character on the first edge. The trickiest part of the implementation is to properly identify these cases. See Exercise 99 for details.

Program 7.2.3 illustrates a fundamental paradigm that we have emphasized throughout this book: Pick an intermediate abstraction (in this case an imaginary machine, the NFA). Build a simulator that makes concrete that abstraction and build a compiler that transforms an instance of the given problem to an instance of the (now-understood) abstraction. This very same basic process is used to compile Java programs into a program for the Java virtual machine (which itself is an abstraction eventually simulated with machine-language programs for particular computers).

---

**Parse method for or operation in NFA constructor**

```java
if ((i = re.indexOf('|')) > 0)
{
    NFAR(from, to, re.substring(0, i));
    NFAR(from, to, re.substring(i+1));
}
```

---

**Diagram**

```
  from  
  /  
R | S 
  
    to 
  
  from  
  
    R 

re.substring(0, i)  re.substring(i+1)
```

---

**Table**

```
+-----------------+-----------------+
|     R           |     S           |
+-----------------+-----------------+
| re.substring(0, i) | re.substring(i+1) |
+-----------------+-----------------+
```
A very common use of REs is for generalized search. That is, rather than asking whether a specific given string is in the language described by the RE, we consider a set of strings and identify which of them is in the language. The original and most common form of this operation is to print out all of the lines in a file with a substring that matches the RE. This operation is commonly known as **grep**, taken from the initial letters in the phrase "generalized regular expression pattern-match." For example, one simple application might involve a database of student grades with one line per student. With grep it is easy to print out the line that has a given student’s grades. Other applications are discussed in Exercise 99. Program 7.2.4 illustrates how to use the NFA class of Program 7.2.2 with the RE-to-NFA constructor of Program 7.2.3 to implement this functionality.
In practice, we always use strings of characters, not bitstrings, and we use various shorthand notations beyond having a wildcard character that matches any character (the . that we have been using throughout). Originally, the generalized in grep referred to REs with parentheses; in modern systems it is used to refer to all sorts of useful extensions. They make RE recognition into a practical tool that you are likely to find yourself using extensively.

Generalized REs. You can find a full definition of what constitutes an RE in Java in Appendix 99. Here, we give brief descriptions of RE extensions, which fall into three main categories.

Program 7.2.5 Generalized RE pattern matching.

```java
public class grep {
    public static void main(String[] args) {
        String s;
        String ab = "abcdefghijklmnopqrstuvwxyz" + "ABCDEFGHIJKLMNOPQRSTUVWXYZ" + "0123456789";
        NFA nfa = new NFA(ab, ".*" + args[0] + ".*");
        In in = new In(args[1]);
        while ((s = in.readLine()) != null)
            if (nfa.simulate(new Tape(s)))
                System.out.println(s);
    }
}
```

This program prints all the lines in the file given as second argument having any substring that matches the RE given as first argument. The examples indicate two common uses: to pull records out of a database and to search for identifiers in code.

% javac grep
% more grades
Alice 92 89 11 10
Bob   99 99 12 15
Carol 68 79 13 13
Dave  78 77 23 12
% java grep Bob grades
Bob   99 99 12 15
% java grep String grep.java
{ String s;
• Expanding the alphabet
• Shorthand notations for the or operation
• Extensions to the closure operation

For brevity, we refer to Java’s generalized regular expressions as generalized REs without drawing finer distinctions. Similar mechanisms are widely defined, in other languages and in other applications, but the exact definition of what constitutes “generalized” is not uniformly applied. For proper usage, the one characteristic that all generalizations should have in common is that every generalized RE should describe a regular language. That is, you can (in principle) translate any generalized RE into a (more cumbersome) standard RE like the ones we have been considering. This restriction is ironic, because it actually says that we are not generalizing REs, just the language that we use to describe regular languages. In due time, we will consider true generalizations.

The alphabet is the most obvious generalization. Java REs are Unicode characters, so there are $2^{16}$ possibilities for each character, not just two. An inherent problem that arises immediately when we extend the alphabet is that we need escape mechanisms to allow us to use characters like . | * ( ) and others both to specify REs and as characters in the language alphabet. You can find details on this issue in Appendix 99.

Having a large number of characters in the alphabet immediately creates the need for shorthand notations for the or operation so that we can specify groups of characters with a few keystrokes:

• The . character matches any character in the alphabet.
• A list or range of characters enclosed in square braces [ ] matches any character in the list or range.
• If the first character within the square braces is the ^ character, the specification refers to the characters not in the list or range.
• Various special characters match a defined set of characters. For example, \s matches any sequence of whitespace characters.

For example, [a-z] represents lower-case letters, [0-9] represents decimal digits, [^aeiou] represents characters that are not vowels, and [A-Z] [a-z]* represents capitalized words like proper names.
The closure operation is often too general to be useful directly in practice. Java REs have the following option for specifying restrictions on the number of repetitions:

- One or more (+)
- Zero or one (?)
- Exactly N ({N})
- Between M and N ({M, N})

For example, \[\text{[^aeiou}\{6\}\] specifies six-letter words that have no lowercase vowels, such as rhythm and syzygy.

Each of these notations is shorthand for a standard RE specification, though the RE might be very long. For example, the dot notation is just shorthand for \((0 \text{ or } 1)\) for the binary alphabet; for Unicode, it is shorthand for a very long string. Similarly a notation like \[^aeiou\] is shorthand for a long sequence of \(\text{or}\) operations for all the other characters, and \[^aeiou\}\{6\}\] is shorthand for six copies of that string. In principle, we could implement a program like Program 7.2.4 for generalized REs by first translating into a long standard RE; in practice, each extension is an opportunity for ingenuity in algorithm design. Perhaps the most effective way to proceed is to raise the level of abstraction of the underlying NFA, for example to use a more general character-matching abstraction than simple equality (see Exercise 99).

You will encounter numerous generalized REs from this point forward. As usual, your task is to study each one, check why the true examples are in the specified language and the false examples are not, and try to understand how the REs do their job. If you are not already convinced, you will soon realize that generalized REs are an effective way to specify sets of strings. How do we make actually make use of them in practical applications? They arise in numerous scenarios, some of which are listed next.
### Validity checking.
When you use the web, you frequently encounter RE recognition. When you type in a date or an account number on a commercial website, it is common for a program to check that your response is in the right format. The analog to Program 7.2.3 for generalized REs is a one-liner in Java, because regular-expression pattern matching is built in to the String class:

```
System.out.println(response.matches(re));
```

Libraries of REs for common checks have sprung up on the web as this type of checking has come into widespread use.

### Searching.
We are always searching. Again, the analog to Program 7.2.4 is easy to develop with Java’s built-in RE tools. We could use the matches in the String class as above, and Java also has a more flexible Pattern class that is like our NFA class: Given a String that is a generalized RE, it creates a Matcher that we can use for searching in the way that we use simulate in Program 7.2.4 (see Exercise 7.2.99). As indicated in the example uses of that program, we can use such a tool for all sorts of searching tasks. You are surely

---

### Some examples of generalized regular expressions

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a-zA-Z-]+@[a-zA-Z.]+...</td>
<td><a href="mailto:XX@yahoo.com">XX@yahoo.com</a></td>
<td>@world.com</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:rs@princeton.edu">rs@princeton.edu</a></td>
<td>nobody@wherever</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:who@whitehouse.gov">who@whitehouse.gov</a></td>
<td><a href="mailto:somebody@.edu">somebody@.edu</a></td>
</tr>
<tr>
<td>[[a-zA-Z-]+@[a-zA-Z.]+...</td>
<td><a href="mailto:XX@yahoo.com">XX@yahoo.com</a></td>
<td>@world.com</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:rs@princeton.edu">rs@princeton.edu</a></td>
<td>nobody@wherever</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:who@whitehouse.gov">who@whitehouse.gov</a></td>
<td><a href="mailto:somebody@.edu">somebody@.edu</a></td>
</tr>
<tr>
<td>[\d+a-zA-Z-]+@[a-zA-Z.]+...</td>
<td><a href="mailto:XX@yahoo.com">XX@yahoo.com</a></td>
<td>@world.com</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:rs@princeton.edu">rs@princeton.edu</a></td>
<td>nobody@wherever</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:who@whitehouse.gov">who@whitehouse.gov</a></td>
<td><a href="mailto:somebody@.edu">somebody@.edu</a></td>
</tr>
<tr>
<td>[\d+a-zA-Z-]+@[a-zA-Z.]+...</td>
<td><a href="mailto:XX@yahoo.com">XX@yahoo.com</a></td>
<td>@world.com</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:rs@princeton.edu">rs@princeton.edu</a></td>
<td>nobody@wherever</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:who@whitehouse.gov">who@whitehouse.gov</a></td>
<td><a href="mailto:somebody@.edu">somebody@.edu</a></td>
</tr>
<tr>
<td>[\d+a-zA-Z-]+@[a-zA-Z.]+...</td>
<td><a href="mailto:XX@yahoo.com">XX@yahoo.com</a></td>
<td>@world.com</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:rs@princeton.edu">rs@princeton.edu</a></td>
<td>nobody@wherever</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:who@whitehouse.gov">who@whitehouse.gov</a></td>
<td><a href="mailto:somebody@.edu">somebody@.edu</a></td>
</tr>
<tr>
<td>[\d+a-zA-Z-]+@[a-zA-Z.]+...</td>
<td><a href="mailto:XX@yahoo.com">XX@yahoo.com</a></td>
<td>@world.com</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:rs@princeton.edu">rs@princeton.edu</a></td>
<td>nobody@wherever</td>
</tr>
<tr>
<td>[[0-9]+\.[0-9]+[0-9]+]</td>
<td><a href="mailto:who@whitehouse.gov">who@whitehouse.gov</a></td>
<td><a href="mailto:somebody@.edu">somebody@.edu</a></td>
</tr>
</tbody>
</table>
familiar with using simple string search as a built in capability in all sorts of applications; with regular expressions you can have searches that are both more flexible and more precise.

[Add some examples here.]

**Program 7.2.6  Harvesting information.**

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class Harvest
{
    public static void main(String[] args)
    {
        In in = new In(args[1]);
        Pattern pattern = Pattern.compile(args[0]);
        Matcher matcher = pattern.matcher(in.readAll());
        while (matcher.find())
        System.out.println(matcher.group());
    }
}
```

This program reads in a URL from the command line and prints out all matching substrings that appear on that web page. It could be used to harvest email addresses, social security numbers, or members of any regular language whatever.

% java Harvester \ [a-z]*@[a-z.]* \ http://www.cs.princeton.edu \ rs@cs.princeton.edu \ wayne@cs.princeton.edu

Harvesting information. The Java Matcher class can tell us not just whether there is a match, but also the substrings that it finds that are in the language described by the RE. Program 7.2.5 is an implementation that uses this tools to print out all the substrings in a file that match a given RE. On reflection, you will see that this program is a powerful and general tool. For example, you could print out all the all the words with hh in them on a web page by typing

```bash
java Harvest " .*hh.* " http://www.xyz.edu
```
or all the Social Security numbers found on another one by typing
All sorts of applications involve searching through large amounts of data, and skill in applying pattern-matching tools can play a critical role. Indeed modern science is awash in data, and skill in applying pattern-matching tools can play an important role in moving forward. Examples are found throughout the sciences, but perhaps the most straightforward example is genomics, where a number of research problems reduce to searching through huge databases of strings in the language \([acgt]^+\) (see Exercise 99).

[Add a genomics example.]

[Add an example from another application.]

[Add a simple example on processing slash-delimited data.]

The story of grep is a satisfying example of the interplay between theory and practice. Kleene’s theorem is a fundamental result that sets the stage for us to consider profound questions about the theory of computation, but it also has served as the basis for the development of practical tools that are indispensable in scientific and commercial computing. Without the fundamental research underlying grep, we simply would not have the tool.

You might think that the existence of the practical tool means that the theory has done its job for us and that we can move on. Nothing could be further from the truth, because practical problems constantly arise that take us right back to the theory. Just to take one example, we can imagine wanting, in a practical situation, to check whether two given REs are equivalent. Perhaps someone is claiming to have developed a more compact way to express a certain validity check than an RE that is in use. On its own terms, writing a program for this task seems challenging, to say the least, but in the context of the theory it is not difficult to do, as you will see if you work Exercise 99.

DFAs have limitations. With all the motivation of the previous section, we can focus again on bitstrings and basic REs and turn back to a fundamental theoretical question: Which formal languages can I write an RE for and which can I not? How do we know where to draw the line between formal languages like Prime and Fermat and the regular languages? There must be languages that we can recognize with a conventional computer or a Java program that
we cannot recognize with a DFA, but how do we know for sure? What is it about DFAs that limits their power? You are likely to be very surprised at the answers to some of these questions: on the one hand, we will see next there are some very simple languages that are not regular; on the other hand, we will see in the next section that there is a simple abstract machine only slightly more powerful than DFAs that can handle any computation that any conventional computer can handle.

The DFA model is very simple, so it is not surprising that it has fundamental limitations. In general, it is easier to understand what a particular machine can do than to identify a limitation shared by all machines. Nevertheless, we will describe a specific language that no DFA can recognize. This discovery foreshadows Section 99 when we discover fundamental limitations on conventional computers.

A nonregular language. Consider the language of all bitstrings that have an equal number of 0s and 1s. This would seem to be at least as simple as many of the languages that we have been considering in this section. Can we write an RE that describes this language? We certainly can imagine a practitioner encountering it. The truth is that this language is not regular, so there is no way to describe it with an RE.

How can we prove such a result? We will use two simple mathematical techniques that you should take the time to understand before reading the proof in the next paragraph. The first is proof by contradiction: to prove that a statement is true, we begin by assuming that it is false. Then we make a step-by-step sequence of logical inferences to arrive at a conclusion that is certainly false. For this conclusion to make sense, the original assumption must be wrong: that is, the statement has to be true. The second proof technique is the pigeonhole principle: suppose that a mailman puts each of $N+1$ items into a mailbox in the mailroom and there are $N$ or fewer mailboxes. Then some
mailbox must have more than one item. If you are not comfortable about these two ideas after thinking about them, see Appendix 99 for some more examples.

So, for the sake of contradiction, let us begin by assuming that we can write down an RE that describes \textit{Equal}. By Kleene’s theorem, we therefore can build a DFA that can recognize the language. Let \( N \) denote the number of states in that DFA. Since it recognizes \textit{Equal}, it will recognize any string with an equal number of 0s and 1s. In particular, it recognizes a string with \( N+1 \) 0s followed by \( N+1 \) 1s. Now, consider the trace of the states that the DFA visits while recognizing this string. Since the DFA has only \( N \) states, the pigeonhole principle tells us that the DFA must revisit one (or more) of the states while reading in the 0s. For example, if \( N \) were 10, the trace might look like the following:

\[
0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1
0-3-5-2-9-7-3-5-2-9-7-3-5-4-2-1-9-6-8-7-6-8-7
\]

Now, find the first repeated state and construct a different input string by removing from the original a 0 for each state transition between the repetitions. In the example above, we would remove five 0s corresponding to the transitions 3–5–2–9–7–3 and we also can deduce that the trace for this string has to look like the following:

\[
0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1
0-3-5-2-9-7-3-5-4-2-1-9-6-8-7-6-8-7
\]

The key observation is that the DFA must end up in the same final state with the second input as it does with the original input. Therefore it accepts the second input. But that input has fewer 0s than 1s. This contradicts our assumption that the DFA recognizes \textit{Equal}, which followed from our assumption that there exists an RE that can describe \textit{Equal}. Thus, we have proven by contradiction that no DFA can accept \textit{Equal}.

This proof is yet another illustration of the importance of Kleene’s theorem. A direct proof that no RE can describe \textit{Equal} or that no NFA can recognize \textit{Equal} is not so easy to develop. The equivalence of REs, DFAs, and
NFAs that is established by Kleene’s theorem means that we can use any one of them to characterize the concept of a regular language.

Showing that just one language is not regular may not seem like much of an accomplishment, but the proof technique applies to many other languages and helps us to understand what characterizes regular languages. Plenty of useful simple languages are not regular.

More important, knowing a simple language that is not regular opens up a critical question for us to consider. What is the simplest way to extend our abstract-machine model to get machines that can recognize Equal? By Kleene’s theorem, even the power of nondeterminism would not help. What will? When we do find such a model, we will need to understand the set of languages its machines can recognize and then to seek limits on the power of its machines. You might expect that we are entering into a process of having to consider a long list of machine models, each with ever-so-slightly more power than the last, but this list actually turns out to be quite short.
Summary. We now have better context for the questions that we mentioned at the beginning of this chapter.

- Are some computers intrinsically more powerful than others?
- What kinds of problems can we solve with a computer?
- What are the limits to what computers can do?

In this section, we have thoroughly addressed these questions for finite automata, which are simple abstract computers that we can precisely define. In addressing these questions, we considered numerous practical applications of these simple abstract models. Moreover, we added two more fundamental questions to the list.

- What is the relationship between computers and languages?
- Are nondeterministic machines more powerful than deterministic ones?

Next, we will consider these questions for more powerful computational models. The models that we need to consider are just slightly more complicated than DFAs and NFAs, but they apply to all known computational devices!
Q&A

Q  What is the closure of the empty set?

A  The empty string: \( \{\}^* = \varepsilon \).

Q  I’m not afraid of the math. Could you provide details on the mathematical representation of DFAs?

A  Sure. A DFA is defined to be a nonnegative integer \( N \) and a triple of functions \( A, S_0, \) and \( S_1 \), where the domain of all the functions is the integers between 0 and \( N - 1 \), the range of \( A \) is the boolean values \( \text{true} \) or \( \text{false} \), and the range of \( S_0 \) and \( S_1 \) is the same as their domain. This definition is just a mathematical way of describing the DFA tabular representation that enables us to give a mathematical description of computation: Each DFA defines a formal language, as follows: a bitstring \( b_0 b_1 b_2 \ldots b_t \) is in the language defined by the DFA if and only if there exists a sequence of states \( p_0 p_1 p_2 \ldots p_t \), such that \( p_0 = 0, S_{b_i}(p_i) = p_{i+1} \) for \( 0 \leq i < t \) and \( A(p_t) = \text{true} \). This formal mathematical description can replace the informal English-language description that we used to introduce the DFA abstraction and can serve as the basis for formal mathematical proofs of the theorems in this section.

Q  How about NFAs?

A  NFAs are actually simpler to define, which is one reason for their appeal. An NFA is a nonnegative integer \( N \) and a triple of functions \( S, S_0, \) and \( S_1 \), where the domain of the functions is the integers between 0 and \( N - 1 \), and the range of the functions is \( \text{sets} \) of integers between 0 and \( N - 1 \). Each NFA defines a formal language, as follows: a bitstring \( b_0 b_1 b_2 \ldots b_t \) is in the language defined by the NFA if and only if there exists a sequence of states \( p_0 p_1 p_2 \ldots p_t \) such that \( p_0 = 0; p_{i+1} \) belongs to \( S(p_i) \) or \( S_{b_i}(p_i) \) for \( 0 \leq i < t \); and \( p_t = 1 \). This mathematical description of nondeterministic computation is perhaps more concrete than thinking about machines that can guess, and it can serve as the basis for formal mathematical proofs about NFAs.

Exercises

7.2.1  Give a brief English description of each of the following REs:
7.2.2 Give an RE that describes the following languages.
• All strings except the empty string.
• Contains at least three consecutive 1s.
• Starts with 0 and has odd length, or starts with 1 and has even length.
• No consecutive 1s.
• Any string except 11 or 11.
• Contains at least two 0s and at most one 1.

7.2.3 What is the closure of the set consisting of the empty string?

7.2.4 Write a Java regular expression to match all lines that contain the 5 vowels (a, e, i, o, u) in order, and no other vowels, eg., abstemious, facetious.

7.2.5 Write a Java regular expression for decimal numbers. A decimal number is a sequence of digits, followed by a period, followed by a sequence of digits. At least one of the digit sequences but be nonempty.

7.2.6 Write a Java regular expression for floating point literals in Java. A floating point literal is a decimal number, followed by e or E, followed optionally by + or -, followed by an integer mantissa.

7.2.7 Write a Java program to read in a list of words and print out those that consist of only lower case letters and have between 7 and 10 letters.

7.2.8 Create a DFA that recognizes that language of all strings with an odd number of a’s and an even number of b’s.

7.2.9 Apply the algorithm in Section XYZ to convert the DFA from the previous exercise into a regular expression.

7.2.10 Given a RE, determine an equivalent NFA.

7.2.11 Given an NFA, determine an equivalent DFA using the recipe described in Section XYZ.
Creative Exercises

7.2.12 Design an DFA that recognizes the language consisting of all bit strings that represent binary numbers that are divisible by 3. For example, 1111 is accepted since 1111 is 15 in decimal, but 1110 is rejected. Hint: use three states. Depending on the input read in so far, your DFA should be in one of three states depending on the remainder when dividing 3 into that number.

7.2.13 A finite state transducer is a DFA whose output is a string of symbols. It is the same as a DFA except that each transition is labeled with an output symbol. After each transition, the transducer output the corresponding symbol. *Bounce filter example.*

7.2.14 Wildcard matcher.

7.2.15 Design a combinatorial lock that opens if it sees the symbols 28-40-51 entered in order.

7.2.16 Write a program that reads in the description of an NFA and creates an DFA that recognizes the same language.

7.2.17 Given a regular language L, show that the set of minimal strings in L is also regular. By minimal, we mean that if x is in L then xy is not in L for any nonempty string y. Answer: modify DFA so that once it leaves an accept state, it always rejects.

7.2.18 Write a program to read in a description of an NFA and an input string and determine whether or not the NFA accepts the input string. Simulate the NFA using a boolean array or deque to avoid the exponential blowup in running time that could occur if you translated the NFA into an DFA first.

7.2.19 Construct a regular expression that describes all binary strings that when interpreted as a binary number are divisible by three. See Exercise XYZ.

7.2.20 Is it possible to construct a regular expression that describes all binary strings that have an equal number of occurrences of 01 and 10 as substrings? Answer: Yes: 0.*0 or 1.*1 or 0* or 1*. It’s true that DFAs
can’t “count”, but in this case the language is equivalent to all bit strings that start and end with the same bit, so no counting is needed.

7.2.21 Show that no DFA can recognize the language ∅, 01 0011 001111 00001111 and so forth.

7.2.22 Show that the back-reference operation is a not-so-regular expression which cannot be constructed from the core regular expression operations. Show that no DFA can recognize the language consisting of ww where w is some string, e.g., beriberi, couscous, but the Java regular expression (.\*:\1 describes this language.

7.2.23 Show that the language containing all bit strings that don’t contain the substring 110 but have a multiple of three 1’s is regular.

7.2.24 Construct an NFA that recognizes the language consisting of all inputs whose third to last character is b.

7.2.25 Apply the subset construction method to create a DFA that is equivalent to the NFA from the previous exercise.

7.2.26 What’s the minimum number of states you need in an NFA that recognizes the language consisting of all inputs whose Kth to the last character is b? What about for a DFA?

Answer: For an NFA, K states suffices, but the subset construction converts the NFA into a DFA with roughly $2^K$ states, and this many states are actually required. Thus, although we can always find a DFA that is equivalent to any NFA, it might be exponentially larger in size.

Exercises to do.

7.2.27 REs with set difference, complement, intersection?

7.2.28 Ambiguity
7.2.29 DFA for 0.*0+1.*1
7.2.30 DFA for .*00101.*
7.2.31 Complement
7.2.32 Build DFA with AND gates for states, etc
7.2.33 KMP DFA
   Solution:
7.2.34 Build a circuit for a coin-change machine.
7.2.35 Implement object-oriented DFA with State class, etc.
7.2.36 Various programs for DFA machine.
7.2.37 Circuit for universal DFA machine.
7.2.38 Universal DFA simulator in TOY.
7.2.39 Write a program to determine the probability that a random bitstring of length N will be recognized by a given DFA.
7.2.40 Change REp to print the match ala Java
7.2.41 DFA equivalence.
7.2.42 RE equivalence.
7.2.43 RE equivalence with {N} is intractable!
7.2.44 Program that parses RE randomly, or right to left
7.2.45 Code to handle parens
7.2.46 Various GRE extensions, showing equivalences
7.2.47 Exception in Java REs (back references)
7.2.48 Java program for BLAST?
7.2.49 Java program for crosswords

DFA for bitstrings whose first bits and last bits are equal
Numerous critical questions in the field of genomics have been reduced to the study of strings over the alphabet \{a, c, t, g\} (genomic sequences) or the alphabet consisting of the capital letters A through I (protein sequences). This sentence belongs in the language section.