The theory of computation begins with a question: What is a computer? It is perhaps a silly question, as everyone knows that this thing I type on is a computer. But these real computers are quite complicated—too much so to allow us to set up a manageable mathematical theory of them directly. Instead we use an idealized computer called a computational model. As with any model in science, a computational model may be accurate in some ways but perhaps not in others. Thus we will use several different computational models, depending on the features we want to focus on. We begin with the simplest model, called the finite state machine or finite automaton.

1.1

FINITE AUTOMATA

Finite automata are good models for computers with an extremely limited amount of memory. What can a computer do with such a small memory? Many useful things! In fact, we interact with such computers all the time, as they lie at the heart of various electromechanical devices.

The controller for an automatic door is one example of such a device. Often found at supermarket entrances and exits, automatic doors swing open when sensing that a person is approaching. An automatic door has a pad in front to
detect the presence of a person about to walk through the doorway. Another pad is located to the rear of the doorway so that the controller can hold the door open long enough for the person to pass all the way through and also so that the door does not strike someone standing behind it as it opens. This configuration is shown in the following figure.

![Figure 1.1: Top view of an automatic door](image)

The controller is in either of two states: "OPEN" or "CLOSED," representing the corresponding condition of the door. As shown in the following figures, there are four possible input conditions: "FRONT" (meaning that a person is standing on the pad in front of the doorway), "REAR" (meaning that a person is standing on the pad to the rear of the doorway), "BOTH" (meaning that people are standing on both pads), and "NEITHER" (meaning that no one is standing on either pad).

![Figure 1.2: State diagram for automatic door controller](image)
1.1 FINITE AUTOMATA

The controller moves from state to state, depending on the input it receives. When in the CLOSED state and receiving input NEITHER or REAR, it remains in the CLOSED state. In addition, if the input BOTH is received, it stays CLOSED because opening the door risks knocking someone over on the rear pad. But if the input FRONT arrives, it moves to the OPEN state. In the OPEN state, if input FRONT, REAR, or BOTH is received, it remains in OPEN. If input NEITHER arrives, it returns to CLOSED.

For example, a controller might start in state CLOSED and receive the series of input signals FRONT, REAR, NEITHER, FRONT, BOTH, NEITHER, REAR, and NEITHER. It then would go through the series of states CLOSED (starting), OPEN, OPEN, CLOSED, OPEN, OPEN, CLOSED, CLOSED, and CLOSED.

Thinking of an automatic door controller as a finite automaton is useful because that suggests standard ways of representation as in Figures 1.2 and 1.3. This controller is a computer that has just a single bit of memory, capable of recording which of the two states the controller is in. Other common devices have controllers with somewhat larger memories. In an elevator controller a state may represent the floor the elevator is on and the inputs might be the signals received from the buttons. This computer might need several bits to keep track of this information. Controllers for various household appliances such as dishwashers and electronic thermostats, as well as parts of digital watches and calculators, are additional examples of computers with limited memories. The design of such devices requires keeping the methodology and terminology of finite automata in mind.

Finite automata and their probabilistic counterpart Markov chains are useful tools when we are attempting to recognize patterns in data. These devices are used in speech processing and in optical character recognition. Markov chains have even been used to model and predict price changes in financial markets.

We will now take a closer look at finite automata from a mathematical perspective. We will develop a precise definition of a finite automaton, terminology for describing and manipulating finite automata, and theoretical results that describe their power and limitations. Besides giving you a clearer understanding of what finite automata are and what they can and cannot do, this theoretical development will allow you to practice and become more comfortable with mathematical definitions, theorems, and proofs in a relatively simple setting.
In beginning to describe the mathematical theory of finite automata, we do so in the abstract, without reference to any particular application. The following figure depicts a finite automaton called $M_1$.

![Figure 1.4](image)

A finite automaton called $M_1$ that has three states

Figure 1.4 is called the **state diagram** of $M_1$. It has three **states**, labeled $q_1$, $q_2$, and $q_3$. The **start state**, $q_1$, is indicated by the arrow pointing at it from nowhere. The **accept state**, $q_2$, is the one with a double circle. The arrows going from one state to another are called **transitions**.

When this automaton receives an input string such as 1101, it processes that string and produces an output. The output is either **accept** or **reject**. We will consider only this yes/no type of output for now to keep things simple. The processing begins in $M_1$’s start state. The automaton receives the symbols from the input string one by one from left to right. After reading each symbol, $M_1$ moves from one state to another along the transition that has that symbol as its label. When it reads the last symbol, $M_1$ produces its output. The output is **accept** if $M_1$ is now in an accept state and **reject** if it is not.

For example, when we feed the input string 1101 to the machine $M_1$ in Figure 1.4, the processing proceeds as follows.

1. Start in state $q_1$.
2. Read 1, follow transition from $q_1$ to $q_2$.
3. Read 1, follow transition from $q_2$ to $q_2$.
4. Read 0, follow transition from $q_2$ to $q_3$.
5. Read 1, follow transition from $q_3$ to $q_2$.
6. **Accept** because $M_1$ is in an accept state $q_2$ at the end of the input.

Experimenting with this machine on a variety of input strings reveals that it accepts the strings 1, 01, 11, and 0101010101. In fact, $M_1$ accepts any string that ends with a 1, as it goes to its accept state $q_2$ whenever it reads the symbol 1. In addition, it accepts strings 100, 0100, 110000, and 0101000000, and any string that ends with an even number of 0s following the last 1. It rejects other strings, such as 0, 10, 101000. Can you describe the language consisting of all strings that $M_1$ accepts? We will do so shortly.
FORMAL DEFINITION OF A FINITE AUTOMATON

In the preceding section we used state diagrams to introduce finite automata. Now we define finite automata formally. Although state diagrams are easier to grasp intuitively, we need the formal definition, too, for two specific reasons.

First, a formal definition is precise. It resolves any uncertainties about what is allowed in a finite automaton. If you were uncertain about whether finite automata were allowed to have 0 accept states or whether they must have exactly one transition exiting every state for each possible input symbol, you could consult the formal definition and verify that the answer is yes in both cases. Second, a formal definition provides notation. Good notation helps you think and express your thoughts clearly.

The language of a formal definition is somewhat arcane, having some similarity to the language of a legal document. Both need to be precise, and every detail must be spelled out.

A finite automaton has several parts. It has a set of states and rules for going from one state to another, depending on the input symbol. It has an input alphabet that indicates the allowed input symbols. It has a start state and a set of accept states. The formal definition says that a finite automaton is a list of those five objects: set of states, input alphabet, rules for moving, start state, and accept states. In mathematical language a list of five elements is often called a 5-tuple. Hence we define a finite automaton to be a 5-tuple consisting of these five parts.

We use something called a transition function, frequently denoted \( \delta \), to define the rules for moving. If the finite automaton has an arrow from a state \( x \) to a state \( y \) labeled with the input symbol 1, that means that, if the automaton is in state \( x \) when it reads a 1, it then moves to state \( y \). We can indicate the same thing with the transition function by saying that \( \delta(x, 1) = y \). This notation is a kind of mathematical shorthand. Putting it all together we arrive at the formal definition of finite automata.

**DEFINITION 1.5**

A finite automaton is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where

1. \( Q \) is a finite set called the **states**,  
2. \( \Sigma \) is a finite set called the **alphabet**,  
3. \( \delta : Q \times \Sigma \rightarrow Q \) is the **transition function**,  
4. \( q_0 \in Q \) is the **start state**, and  
5. \( F \subseteq Q \) is the **set of accept states**.\(^1\)

---

\(^1\)Refer back to page 7 if you are uncertain about the meaning of \( \delta : Q \times \Sigma \rightarrow Q \).

\(^2\)Accept states sometimes are called **final states**.
The formal definition precisely describes what we mean by a finite automaton. For example, returning to the earlier question of whether \(0\) accept states is allowable, you can see that setting \(F\) to be the empty set \(\emptyset\) yields \(0\) accept states, which is allowable. Furthermore, the transition function \(\delta\) specifies exactly one next state for each possible combination of a state and an input symbol. That answers our other question affirmatively, showing that exactly one transition arrow exits every state for each possible input symbol.

We can use the notation of the formal definition to describe individual finite automata by specifying each of the five parts listed in Definition 1.5. For example, let’s return to the finite automaton \(M_1\) we discussed earlier, redrawn here for convenience.

![Diagram of finite automaton](image)

**Figure 1.6**
The finite automaton \(M_1\)

We can describe \(M_1\) formally by writing \(M_1 = (Q, \Sigma, \delta, q_1, F)\), where

1. \(Q = \{q_1, q_2, q_3\}\),
2. \(\Sigma = \{0, 1\}\),
3. \(\delta\) is described as

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_1)</td>
<td>(q_1)</td>
<td>(q_2)</td>
</tr>
<tr>
<td>(q_2)</td>
<td>(q_3)</td>
<td>(q_2)</td>
</tr>
<tr>
<td>(q_3)</td>
<td>(q_2)</td>
<td>(q_2)</td>
</tr>
</tbody>
</table>

4. \(q_1\) is the start state, and
5. \(F = \{q_2\}\).

If \(A\) is the set of all strings that machine \(M\) accepts, we say that \(A\) is the **language of machine** \(M\) and write \(L(M) = A\). We say that \(M\) **recognizes** \(A\) or that \(M\) **accepts** \(A\). Because the term *accept* has different meanings when we refer to machines accepting strings and machines accepting languages, we prefer the term *recognize* for languages in order to avoid confusion.

A machine may accept several strings, but it always recognizes only one language. If the machine accepts no strings, it still recognizes one language—namely, the empty language \(\emptyset\).
In our example, let
\[ A = \{ w \mid w \text{ contains at least one } 1 \text{ and} \] 
an even number of 0s follow the last 1}. 

Then \( L(M_1) = A \), or equivalently, \( M_1 \) recognizes \( A \).

**EXAMPLES OF FINITE AUTOMATA**

**EXAMPLE 1.7**

Here is the state diagram of finite automaton \( M_2 \).

![State diagram of \( M_2 \)](image)

**FIGURE 1.8**
State diagram of the two-state finite automaton \( M_2 \)

In the formal description \( M_2 = (\{ q_1, q_2 \}, \{ 0, 1 \}, \delta, q_1, \{ q_2 \}) \). The transition function \( \delta \) is

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_1 )</td>
<td>( q_1 )</td>
<td>( q_2 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( q_1 )</td>
<td>( q_2 )</td>
</tr>
</tbody>
</table>

Remember that the state diagram of \( M_2 \) and the formal description of \( M_2 \) contain the same information, only in different form. You can always go from one to the other if necessary.

A good way to begin understanding any machine is to try it on some sample input strings. When you do these “experiments” to see how the machine is working, its method of functioning often becomes apparent. On the sample string 1101 the machine \( M_2 \) starts in its start state \( q_1 \) and proceeds first to state \( q_2 \) after reading the first 1, and then to states \( q_2, q_1, \) and \( q_2 \) after reading 1, 0, and 1. The string is accepted because \( q_2 \) is an accept state. But string 110 leaves \( M_2 \) in state \( q_1 \), so it is rejected. After trying a few more examples, you would see that \( M_2 \) accepts all strings that end in a 1. Thus \( L(M_2) = \{ w \mid w \text{ ends in } 1 \} \).
EXAMPLE 1.9

Consider the finite automaton $M_3$.

![State diagram of the two-state finite automaton $M_3$](image)

**FIGURE 1.10**
State diagram of the two-state finite automaton $M_3$

Machine $M_3$ is similar to $M_2$ except for the location of the accept state. As usual, the machine accepts all strings that leave it in an accept state when it has finished reading. Note that, because the start state is also an accept state, $M_3$ accepts the empty string $\varepsilon$. As soon as a machine begins reading the empty string it is at the end, so if the start state is an accept state, $\varepsilon$ is accepted. In addition to the empty string, this machine accepts any string ending with a 0. Here,

$$L(M_3) = \{ w \mid w \text{ is the empty string } \varepsilon \text{ or ends in a } 0 \}.$$

EXAMPLE 1.11

The following figure shows a five-state machine $M_4$

![Finite automaton $M_4$](image)

**FIGURE 1.12**
Finite automaton $M_4$
Machine $M_4$ has two accept states, $q_1$ and $r_1$, and operates over the alphabet $\Sigma = \{a, b\}$. Some experimentation shows that it accepts strings $a$, $b$, $aa$, $bb$, and $bab$, but not strings $ab$, $ba$, or $bbba$. This machine begins in state $s$, and after it reads the first symbol in the input, it goes either left into the $q$ states or right into the $r$ states. In both cases it can never return to the start state (in contrast to the previous examples), as it has no way to get from any other state back to $s$. If the first symbol in the input string is $a$, then it goes left and accepts when the string ends with an $a$. Similarly, if the first symbol is a $b$, the machine goes right and accepts when the string ends in $b$. So $M_4$ accepts all strings that start and end with a $a$ or that start and end with b. In other words, $M_4$ accepts strings that start and end with the same symbol.

**Example 1.13**

Figure 1.14 shows machine $M_5$, which has a four-symbol input alphabet, $\Sigma = \{(\text{RESET}), 0, 1, 2\}$. We treat $(\text{RESET})$ as a single symbol.

**Figure 1.14**

Finite automaton $M_5$

Machine $M_5$ keeps a running count of the sum of the numerical input symbols it reads, modulo 3. Every time it receives the $(\text{RESET})$ symbol it resets the count to 0. It accepts if the sum is 0, modulo 3, or in other words, if the sum is a multiple of 3.

Describing a finite automaton by state diagram is not possible in some cases. That may occur when the diagram would be too big to draw or if, as in this example, the description depends on some unspecified parameter. In these cases we resort to a formal description to specify the machine.
EXAMPLE 1.15

Consider a generalization of Example 1.13, using the same four-symbol alphabet $\Sigma$. For each $i \geq 1$ let $A_i$ be the language of all strings where the sum of the numbers is a multiple of $i$, except that the sum is reset to 0 whenever the symbol $\langle \text{RESET} \rangle$ appears. For each $A_i$ we give a finite automaton $B_i$, recognizing $A_i$. We describe the machine $B_i$ formally as follows: $B_i = (Q_i, \Sigma, \delta_i, q_0, \{q_0\})$, where $Q_i$ is the set of $i$ states \{q_0, q_1, q_2, \ldots, q_{i-1}\}, and we design the transition function $\delta_i$ so that for each $j$, if $B_i$ is in $q_j$, the running sum is $j$, modulo $i$. For each $q_j$ let

\begin{align*}
\delta_i(q_j, 0) &= q_j, \\
\delta_i(q_j, 1) &= q_k, \text{ where } k = j + 1 \text{ modulo } i, \\
\delta_i(q_j, 2) &= q_k, \text{ where } k = j + 2 \text{ modulo } i, \text{ and} \\
\delta_i(q_j, \langle \text{RESET} \rangle) &= q_0.
\end{align*}

FORMAL DEFINITION OF COMPUTATION

So far we have described finite automata informally, using state diagrams, and with a formal definition, as a 5-tuple. The informal description is easier to grasp at first, but the formal definition is useful for making the notion precise, resolving any ambiguities that may have occurred in the informal description. Next we do the same for a finite automaton’s computation. We already have an informal idea of the way it computes, and we now formalize it mathematically.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each $w_i$ is a member of the alphabet $\Sigma$. Then $M$ accepts $w$ if a sequence of states $r_0, r_1, \ldots, r_n$ in $Q$ exists with three conditions:

1. $r_0 = q_0$,  
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, \text{ for } i = 0, \ldots, n - 1, \text{ and}  
3. $r_n \in F$.

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state. We say that $M$ recognizes language $A$ if $A = \{w | M$ accepts $w\}$.

DEFINITION 1.16

A language is called a \textit{regular language} if some finite automaton recognizes it.
EXAMPLE 1.17

Take machine $M_5$ from Example 1.13. Let $w$ be the string

$$10\langle\text{RESET}\rangle 22\langle\text{RESET}\rangle 012$$

Then $M_5$ accepts $w$ according to the formal definition of computation because the sequence of states it enters when computing on $w$ is

$$q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0,$$

which satisfies the three conditions. The language of $M_5$ is

$$L(M_5) = \{w | \text{the sum of the symbols in } w \text{ is 0 modulo 3, except that } \langle\text{RESET}\rangle \text{ resets the count to 0}\}.$$ 

As $M_5$ recognizes this language, it is a regular language.

DESIGNING FINITE AUTOMATA

Whether it be of automaton or artwork, design is a creative process. As such it cannot be reduced to a simple recipe or formula. However, you might find a particular approach helpful when designing various types of automata. That is, put yourself in the place of the machine you are trying to design and then see how you would go about performing the machine’s task. Pretending that you are the machine is a psychological trick that helps engage your whole mind in the design process.

Let’s design a finite automaton using the “reader as automaton” method just described. Suppose that you are given some language and want to design a finite automaton that recognizes it. Pretending to be the automaton, you receive an input string and must determine whether it is a member of the language the automaton is supposed to recognize. You get to see the symbols in the string one by one. After each symbol you must decide whether the string seen so far is in the language. The reason is that you, like the machine, don’t know when the end of the string is coming, so you must always be ready with the answer.

First, in order to make these decisions, you have to figure out what you need to remember about the string as you are reading it. Why not simply remember all you have seen? Bear in mind that you are pretending to be a finite automaton and that this type of machine has only a finite number of states, which means a finite memory. Imagine that the input is extremely long—say, from here to the moon—so that you could not possibly remember the entire thing. You have a finite memory—say, a single sheet of paper—which has a limited storage capacity. Fortunately, for many languages you don’t need to remember the entire input. You need to remember only certain crucial information. Exactly which information is crucial depends on the particular language considered.

For example, suppose that the alphabet is \{0,1\} and that the language consists of all strings with an odd number of 1s. You want to construct a finite automaton $E_1$ to recognize this language. Pretending to be the automaton, you start getting
an input string of 0s and 1s symbol by symbol. Do you need to remember the entire string seen so far in order to determine whether the number of 1s is odd? Of course not. Simply remember whether the number of 1s seen so far is even or odd and keep track of this information as you read new symbols. If you read a 1, flip the answer, but if you read a 0, leave the answer as is.

But how does this help you design $E_1$? Once you have determined the necessary information to remember about the string as it is being read, you represent this information as a finite list of possibilities. In this instance, the possibilities would be

1. even so far, and
2. odd so far.

Then you assign a state to each of the possibilities. These are the states of $E_1$, as shown here.

![Figure 1.18](image)

**Figure 1.18**
The two states $q_{\text{even}}$ and $q_{\text{odd}}$

Next, you assign the transitions by seeing how to go from one possibility to another upon reading a symbol. So, if state $q_{\text{even}}$ represents the even possibility and state $q_{\text{odd}}$ represents the odd possibility, you would set the transitions to flip state on a 1 and stay put on a 0, as shown here.

![Figure 1.19](image)

**Figure 1.19**
Transitions telling how the possibilities rearrange

Next, you set the start state to be the state corresponding to the possibility associated with having seen 0 symbols so far (the empty string $\varepsilon$). In this case the start state corresponds to state $q_{\text{even}}$ because 0 is an even number. Last, set the accept states to be those corresponding to possibilities where you want to accept the input string. Set $q_{\text{odd}}$ to be an accept state because you want to accept when
you have seen an odd number of 1s. These additions are shown in the following figure.

![Diagram](image)

**Figure 1.20**
Adding the start and accept states

**Example 1.21**
This example shows how to design a finite automaton $E_2$ to recognize the regular language of all strings that contain the string 001 as a substring. For example, 0010, 1001, 001, and 11111110011111 are all in the language, but 11 and 0000 are not. How would you recognize this language if you were pretending to be $E_2$? As symbols come in, you would initially skip over all 1s. If you come to a 0, then you note that you may have just seen the first of the three symbols in the pattern 001 you are seeking. If at this point you see a 1, there were too few 0s, so you go back to skipping over 1s. But if you see a 0 at that point, you should remember that you have just seen two symbols of the pattern. Now you simply need to continue scanning until you see a 1. If you find it, remember that you succeeded in finding the pattern and continue reading the input string until you get to the end.

So there are four possibilities: You

1. haven’t just seen any symbols of the pattern,
2. have just seen a 0,
3. have just seen 00, or
4. have seen the entire pattern 001.

Assign the states $q$, $q_0$, $q_{00}$, and $q_{001}$ to these possibilities. You can assign the transitions by observing that from $q$ reading a 1 you stay in $q$, but reading a 0 you move to $q_0$. In $q_0$ reading a 1 you return to $q$, but reading a 0 you move to $q_{00}$. In $q_{00}$, reading a 1 you move to $q_{001}$, but reading a 0 leaves you in $q_{00}$. Finally, in $q_{001}$ reading a 0 or a 1 leaves you in $q_{001}$. The start state is $q$, and the only accept state is $q_{001}$, as shown in Figure 1.22.
THE REGULAR OPERATIONS

In the preceding two sections we introduced and defined finite automata and regular languages. We now begin to investigate their properties. Doing so will help develop a toolbox of techniques to use when you design automata to recognize particular languages. The toolbox also will include ways of proving that certain other languages are nonregular (i.e., beyond the capability of finite automata).

In arithmetic, the basic objects are numbers and the tools are operations for manipulating them, such as $+$ and $\times$. In the theory of computation the objects are languages and the tools include operations specifically designed for manipulating them. We define three operations on languages, called the regular operations, and use them to study properties of the regular languages.

**DEFINITION 1.23**

Let $A$ and $B$ be languages. We define the regular operations union, concatenation, and star as follows.

- **Union:** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$.
- **Star:** $A^* = \{x_1x_2\ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$.

You are already familiar with the union operation. It simply takes all the strings in both $A$ and $B$ and lumps them together into one language.

The concatenation operation is a little trickier. It attaches a string from $A$ in front of a string from $B$ in all possible ways to get the strings in the new language.

The star operation is a bit different from the other two because it applies to a single language rather than to two different languages. That is, the star operation is a unary operation instead of a binary operation. It works by attaching any number of strings in $A$ together to get a string in the new language. Because
“any number” includes 0 as a possibility, the empty string $\varepsilon$ is always a member of $A^*$, no matter what $A$ is.

**Example 1.24**

Let the alphabet $\Sigma$ be the standard 26 letters \{a, b, \ldots, z\}. If $A = \{\text{good, bad}\}$ and $B = \{\text{boy, girl}\}$, then

\[
A \cup B = \{\text{good, bad, boy, girl}\},
\]

\[
A \circ B = \{\text{goodboy, goodgirl, badboy, badgirl}\}, \text{ and}
\]

\[
A^* = \{\varepsilon, \text{good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, goodgoodbad, goodbadgood, goodbadbad, \ldots}\}.
\]

Let $\mathcal{N} = \{1, 2, 3, \ldots\}$ be the set of natural numbers. When we say that $\mathcal{N}$ is **closed under multiplication** we mean that, for any $x$ and $y$ in $\mathcal{N}$, the product $x \times y$ also is in $\mathcal{N}$. In contrast $\mathcal{N}$ is not closed under division, as 1 and 2 are in $\mathcal{N}$ but 1/2 is not. Generally speaking, a collection of objects is **closed** under some operation if applying that operation to members of the collection returns an object still in the collection. We show that the collection of regular languages is closed under all three of the regular operations. In Section 1.3 we show that these are useful tools for manipulating regular languages and understanding the power of finite automata. We begin with the union operation.

**Theorem 1.25**

The class of regular languages is closed under the union operation.

In other words, if $A_1$ and $A_2$ are regular languages, so is $A_1 \cup A_2$.

**Proof Idea**

We have regular languages $A_1$ and $A_2$ and want to show that $A_1 \cup A_2$ also is regular. Because $A_1$ and $A_2$ are regular, we know that some finite automaton $M_1$ recognizes $A_1$ and some finite automaton $M_2$ recognizes $A_2$. To prove that $A_1 \cup A_2$ is regular we demonstrate a finite automaton, call it $M$, that recognizes $A_1 \cup A_2$.

This is a proof by construction. We construct $M$ from $M_1$ and $M_2$. Machine $M$ must accept its input exactly when either $M_1$ or $M_2$ would accept it in order to recognize the union language. It works by simulating both $M_1$ and $M_2$ and accepting if either of the simulations accept.

How can we make machine $M$ simulate $M_1$ and $M_2$? Perhaps it first simulates $M_1$ on the input and then simulates $M_2$ on the input. But we must be careful here! Once the symbols of the input have been read and used to simulate $M_1$, we can’t “rewind the input tape” to try the simulation on $M_2$. We need another approach.
Pretend that you are \( M \). As the input symbols arrive one by one, you simulate both \( M_1 \) and \( M_2 \) simultaneously. That way only one pass through the input is necessary. But can you keep track of both simulations with finite memory? All you need to remember is the state that each machine would be in if it had read up to this point in the input. Therefore you need to remember a pair of states. How many possible pairs are there? If \( M_1 \) has \( k_1 \) states and \( M_2 \) has \( k_2 \) states, the number of pairs of states, one from \( M_1 \) and the other from \( M_2 \), is the product \( k_1 \times k_2 \). This product will be the number of states in \( M \) one for each pair. The transitions of \( M \) go from pair to pair, updating the current state for both \( M_1 \) and \( M_2 \). The accept states of \( M \) are those pairs wherein either \( M_1 \) or \( M_2 \) is in an accept state.

**Proof**

Let \( M_1 \) recognize \( A_1 \), where \( M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1) \), and
\( M_2 \) recognize \( A_2 \), where \( M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) \).

Construct \( M \) to recognize \( A_1 \cup A_2 \), where \( M = (Q, \Sigma, \delta, q_0, F) \).

1. \( Q = \{(r_1, r_2) | r_1 \in Q_1 \text{ and } r_2 \in Q_2 \} \).
   This set is the *Cartesian product* of sets \( Q_1 \) and \( Q_2 \) and is written \( Q_1 \times Q_2 \).
   It is the set of all pairs of states, the first from \( Q_1 \) and the second from \( Q_2 \).

2. \( \Sigma \), the alphabet, is the same as in \( M_1 \) and \( M_2 \). In this theorem and in all subsequent similar theorems, we assume for simplicity that both \( M_1 \) and \( M_2 \) have the same input alphabet \( \Sigma \). The theorem remains true if they have different alphabets, \( \Sigma_1 \) and \( \Sigma_2 \). We would then modify the proof to let \( \Sigma = \Sigma_1 \cup \Sigma_2 \).

3. \( \delta \), the transition function, is defined as follows. For each \( (r_1, r_2) \in Q \) and each \( a \in \Sigma \), let
   \[
   \delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)).
   \]
   Hence \( \delta \) gets a state of \( M \) (which actually is a pair of states from \( M_1 \) and \( M_2 \)), together with an input symbol, and returns \( M \)'s next state.

4. \( q_0 \) is the pair \( (q_1, q_2) \).

5. \( F \) is the set of pairs in which either member is an accept state of \( M_1 \) or \( M_2 \).
   We can write it as
   \[
   F = \{(r_1, r_2) | r_1 \in F_1 \text{ or } r_2 \in F_2 \}.
   \]
   This expression is the same as \( F = (F_1 \times Q_2) \cup (Q_1 \times F_2) \). (Note that it is *not* the same as \( F = F_1 \times F_2 \). What would that give us instead?\(^3\))

\(^3\) This expression would define \( M \)'s accept states to be those for which *both* members of the pair are accept states. In this case \( M \) would accept a string only if both \( M_1 \) and \( M_2 \) accept it, so the resulting language would be the *intersection* and not the union. In fact, this result proves that the class of regular languages is closed under intersection.
This concludes the construction of the finite automaton \( M \) that recognizes the union of \( A_1 \) and \( A_2 \). This construction is fairly simple, and thus its correctness is evident from the strategy described in the proof idea. More complicated constructions require additional discussion to prove correctness. A formal correctness proof for a construction of this type usually proceeds by induction. For an example of a construction proved correct, see the proof of Theorem 1.54. Most of the constructions that you will encounter in this course are fairly simple and so do not require a formal correctness proof.

We have just shown that the union of two regular languages is regular, thereby proving that the class of regular languages is closed under the union operation. We now turn to the concatenation operation and attempt to show that the class of regular languages is closed under that operation, too.

**THEOREM 1.26**

The class of regular languages is closed under the concatenation operation.

In other words, if \( A_1 \) and \( A_2 \) are regular languages then so is \( A_1 \circ A_2 \).

To prove this theorem let's try something along the lines of the proof of the union case. As before, we can start with finite automata \( M_1 \) and \( M_2 \) recognizing the regular languages \( A_1 \) and \( A_2 \). But now, instead of constructing automaton \( M \) to accept its input if either \( M_1 \) or \( M_2 \) accept, it must accept if its input can be broken into two pieces, where \( M_1 \) accepts the first piece and \( M_2 \) accepts the second piece. The problem is that \( M \) doesn't know where to break its input (i.e., where the first part ends and the second begins). To solve this problem we introduce a new technique called nondeterminism.

**1.2 NONDETERMINISM**

Nondeterminism is a useful concept that has had great impact on the theory of computation. So far in our discussion, every step of a computation follows in a unique way from the preceding step. When the machine is in a given state and reads the next input symbol, we know what the next state will be—it is determined. We call this **deterministic** computation. In a **nondeterministic** machine, several choices may exist for the next state at any point.

Nondeterminism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton. As Figure 1.27 shows, nondeterministic finite automata may have additional features.
The difference between a deterministic finite automaton, abbreviated DFA, and a nondeterministic finite automaton, abbreviated NFA, is immediately apparent. First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. The nondeterministic automaton shown in Figure 1.27 violates that rule. State $q_1$ has one exiting arrow for 0, but it has two for 1; $q_2$ has one arrow for 0, but it has none for 1. In an NFA a state may have zero, one, or many exiting arrows for each alphabet symbol.

Second, in a DFA, labels on the transition arrows are symbols from the alphabet. This NFA has an arrow with the label $\varepsilon$. In general, an NFA may have arrows labeled with members of the alphabet or $\varepsilon$. Zero, one, or many arrows may exit from each state with the label $\varepsilon$.

How does an NFA compute? Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. For example, say that we are in state $q_1$ in NFA $N_1$ and that the next input symbol is a 1. After reading that symbol, the machine splits into multiple copies of itself and follows all the possibilities in parallel. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol doesn’t appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it. Finally, if any one of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

If a state with an $\varepsilon$ symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple copies, one following each of the exiting $\varepsilon$-labeled arrows and one staying at the current state. Then the machine proceeds nondeterministically as before.

Nondeterminism may be viewed as a kind of parallel computation wherein multiple independent “processes” or “threads” can be running concurrently. When the NFA splits to follow several choices, that corresponds to a process “forking” into several children, each proceeding separately. If at least one of these processes accepts, then the entire computation accepts.

Another way to think of a nondeterministic computation is as a tree of possibilities. The root of the tree corresponds to the start of the computation. Every branching point in the tree corresponds to a point in the computation at which the machine has multiple choices. The machine accepts if at least one of the computation branches ends in an accept state, as shown in Figure 1.28.
Let’s consider some sample runs of the NFA $N_1$ shown in Figure 1.27. The computation of $N_1$ on input 010110 is depicted in the following figure.
On input 010110 start in the start state $q_1$ and read the first symbol 0. From $q_1$ there is only one place to go on a 0—namely, back to $q_1$, so remain there. Next read the second symbol 1. In $q_1$ on a 1 there are two choices: either stay in $q_1$ or move to $q_2$. Nondeterministically, the machine splits in two to follow each choice. Keep track of the possibilities by placing a finger on each state where a machine could be. So you now have fingers on states $q_1$ and $q_2$. An $\varepsilon$ arrow exits state $q_2$ so the machine splits again; keep one finger on $q_2$, and move the other to $q_3$. You now have fingers on $q_1$, $q_2$, and $q_3$.

When the third symbol 0 is read, take each finger in turn. Keep the finger on $q_1$ in place, move the finger on $q_2$ to $q_3$, and remove the finger that has been on $q_3$. That last finger had no 0 arrow to follow and corresponds to a process that simply “dies.” At this point you have fingers on states $q_1$ and $q_3$.

When the fourth symbol 1 is read, split the finger on $q_1$ into fingers on states $q_1$ and $q_2$, then further split the finger on $q_2$ to follow the $\varepsilon$ arrow to $q_3$, and move the finger that was on $q_3$ to $q_4$. You now have a finger on each of the four states.

When the fifth symbol 1 is read, the fingers on $q_1$ and $q_3$ result in fingers on states $q_1$, $q_2$, $q_3$, and $q_4$, as you saw with the fourth symbol. The finger on state $q_2$ is removed. The finger that was on $q_4$ stays on $q_4$. Now you have two fingers on $q_4$, so remove one, because you only need to remember that $q_4$ is a possible state at this point, not that it is possible for multiple reasons.

When the sixth and final symbol 0 is read, keep the finger on $q_1$ in place, move the one on $q_2$ to $q_3$, remove the one that was on $q_3$, and leave the one on $q_4$ in place. You are now at the end of the string, and you accept if some finger is on an accept state. You have fingers on states $q_1$, $q_3$, and $q_4$, and as $q_4$ is an accept state, $N_1$ accepts this string.

What does $N_1$ do on input 010? Start with a finger on $q_1$. After reading the 0 you still have a finger only on $q_1$, but after the 1 there are fingers on $q_1$, $q_2$, and $q_3$ (don’t forget the $\varepsilon$ arrow). After the third symbol 0, remove the finger on $q_3$, move the finger on $q_2$ to $q_3$, and leave the finger on $q_1$ where it is. At this point you are at the end of the input, and as no finger is on an accept state, $N_1$ rejects this input.

By continuing to experiment in this way, you will see that $N_1$ accepts all strings that contain either 101 or 11 as a substring.

Nondeterministic finite automata are useful in several respects. As we will show, every NFA can be converted into an equivalent DFA, and constructing NFAs is sometimes easier than directly constructing DFAs. An NFA may be much smaller than its deterministic counterpart, or its functioning may be easier to understand. Nondeterminism in finite automata is also a good introduction to nondeterminism in more powerful computational models because finite automata are especially easy to understand. Now we turn to several examples of NFAs.
Let $A$ be the language consisting of all strings over $\{0,1\}$ containing a 1 in the third position from the end (e.g., $000100$ is in $A$ but $0011$ is not). The following four-state NFA $N_2$ recognizes $A$.

![Diagram of NFA N2](image)

**Figure 1.31**
The NFA $N_2$ recognizing $A$

One good way to view the computation of this NFA is to say that it stays in the start state $q_1$ until it "guesses" that it is three places from the end. At that point, if the input symbol is a 1, it branches to state $q_2$ and uses $q_3$ and $q_4$ to "check" on whether its guess was correct.

As mentioned, every NFA can be converted into an equivalent DFA, but sometimes that DFA may have many more states. The smallest DFA for $A$ contains eight states. Furthermore, understanding the functioning of the NFA is much easier, as you may see by examining the following figure for the DFA.

![Diagram of DFA](image)

**Figure 1.32**
A DFA recognizing $A$

Suppose that we added $\varepsilon$ to the labels on the arrows going from $q_2$ to $q_3$ and from $q_3$ to $q_4$ in machine $N_2$ in Figure 1.31. So both arrows would then have the label $0, 1, \varepsilon$ instead of just $0, 1$. What language would $N_2$ recognize with this modification? Try modifying the DFA in Figure 1.32 to recognize that language.
Consider the following NFA $N_3$ that has an input alphabet \{0\} consisting of a single symbol. An alphabet containing only one symbol is called a unary alphabet.

![Diagram of NFA $N_3$]

**Figure 1.34**
The NFA $N_3$

This machine demonstrates the convenience of having $\epsilon$ arrows. It accepts all strings of the form $0^k$ where $k$ is a multiple of 2 or 3. (Remember that the superscript denotes repetition, not numerical exponentiation.) For example, $N_3$ accepts the strings $\epsilon$, 00, 000, 0000, and 000000, but not 0 or 00000.

Think of the machine operating by initially guessing whether to test for a multiple of 2 or a multiple of 3 by branching into either the top loop or the bottom loop and then checking whether its guess was correct. Of course, we could replace this machine by one that doesn't have $\epsilon$ arrows or even any nondeterminism at all, but the machine shown is the easiest one to understand for this language.

**Example 1.35**

We give another example of an NFA in the following figure. Practice with it to satisfy yourself that it accepts the strings $\epsilon$, $a$, $baba$, and $baa$, but that it doesn't accept the strings $b$, $bb$, and $babba$. Later we use this machine to illustrate the procedure for converting NFAs to DFAs.
1.2 NONDETERMINISM

**Diagram 1.36**
The NFA $N_4$

---

**Formal Definition of a Nondeterministic Finite Automaton**

The formal definition of a nondeterministic finite automaton is similar to that of a deterministic finite automaton. Both have states, an input alphabet, a transition function, a start state, and a collection of accept states. However, they differ in one essential way: in the type of transition function. In a DFA the transition function takes a state and an input symbol and produces the next state. In an NFA the transition function takes a state and an input symbol or the empty string and produces the set of possible next states. In order to write the formal definition, we need to set up some additional notation. For any set $Q$ we write $\mathcal{P}(Q)$ to be the collection of all subsets of $Q$. Here $\mathcal{P}(Q)$ is called the **power set** of $Q$. For any alphabet $\Sigma$ we write $\Sigma_e$ to be $\Sigma \cup \{\varepsilon\}$. Now we can write the formal description of the type of the transition function in an NFA as $\delta: Q \times \Sigma_e \rightarrow \mathcal{P}(Q)$.

---

**Definition 1.37**

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

1. $Q$ is a finite set of states,
2. $\Sigma$ is a finite alphabet,
3. $\delta: Q \times \Sigma_e \rightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.
Recall the NFA $N_1$:

The formal description of $N_1$ is $(Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0, 1\}$,
3. $\delta$ is given as

   \[
   \begin{array}{c|ccc}
   & 0 & 1 & \varepsilon \\
   q_1 & \{q_1\} & \{q_1, q_2\} & \emptyset \\
   q_2 & \{q_3\} & \emptyset & \{q_3\} \\
   q_3 & \emptyset & \{q_4\} & \emptyset \\
   q_4 & \{q_4\} & \{q_4\} & \emptyset \\
   \end{array}
   \]

4. $q_1$ is the start state, and
5. $F = \{q_4\}$.

The formal definition of computation for an NFA is similar to that for a DFA. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and $w$ a string over the alphabet $\Sigma$. Then we say that $N$ accepts $w$ if we can write $w$ as $w = y_1 y_2 \cdots y_m$, where each $y_i$ is a member of $\Sigma$ and a sequence of states $r_0, r_1, \ldots, r_m$ exists in $Q$ with three conditions:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, \ldots, m - 1$, and
3. $r_m \in F$.

Condition 1 says that the machine starts out in the start state. Condition 2 says that state $r_{i+1}$ is one of the allowable next states when $N$ is in state $r_i$ and reading $y_{i+1}$. Observe that $\delta(r_i, y_{i+1})$ is the set of allowable next states and so we say that $r_{i+1}$ is a member of that set. Finally, condition 3 says that the machine accepts its input if the last state is an accept state.

**EQUIVALENCE OF NFAS AND DFAS**

Deterministic and nondeterministic finite automata recognize the same class of languages. Such equivalence is both surprising and useful. It is surprising because NFAs appear to have more power than DFAs, so we might expect that NFAs recognize more languages. It is useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language.

Say that two machines are equivalent if they recognize the same language.
1.2 NONDETERMINISM

THEOREM 1.39

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

PROOF IDEA If a language is recognized by an NFA, then we must show the existence of a DFA that also recognizes it. The idea is to convert the NFA into an equivalent DFA that simulates the NFA.

Recall the “reader as automaton” strategy for designing finite automata. How would you simulate the NFA if you were pretending to be a DFA? What do you need to keep track of as the input string is processed? In the examples of NFAs you kept track of the various branches of the computation by placing a finger on each state that could be active at given points in the input. You updated the simulation by moving, adding, and removing fingers according to the way the NFA operates. All you needed to keep track of was the set of states having fingers on them.

If \( k \) is the number of states of the NFA, it has \( 2^k \) subsets of states. Each subset corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have \( 2^k \) states. Now we need to figure out which will be the start state and accept states of the DFA, and what will be its transition function. We can discuss this more easily after setting up some formal notation.

PROOF Let \( N = (Q, \Sigma, \delta, q_0, F) \) be the NFA recognizing some language \( A \). We construct a DFA \( M = (Q', \Sigma, \delta', q'_0, F') \) recognizing \( A \). Before doing the full construction, let's first consider the easier case wherein \( N \) has no \( \varepsilon \) arrows. Later we take the \( \varepsilon \) arrows into account.

1. \( Q' = \mathcal{P}(Q) \).
   Every state of \( M \) is a set of states of \( N \). Recall that \( \mathcal{P}(Q) \) is the set of subsets of \( Q \).

2. For \( R \in Q' \) and \( a \in \Sigma \) let \( \delta'(R, a) = \{ q \in Q | q \in \delta(r, a) \text{ for some } r \in R \} \).
   If \( R \) is a state of \( M \), it is also a set of states of \( N \). When \( M \) reads a symbol \( a \) in state \( R \), it shows where \( a \) takes each state in \( R \). Because each state may go to a set of states, we take the union of all these sets. Another way to write this expression is
   \[
   \delta'(R, a) = \bigcup_{r \in R} \delta(r, a). \tag{4}
   \]

3. \( q'_0 = \{ q_0 \} \).
   \( M \) starts in the state corresponding to the collection containing just the start state of \( N \).

4. \( F' = \{ R \in Q' | R \text{ contains an accept state of } N \} \).
   The machine \( M \) accepts if one of the possible states that \( N \) could be in at this point is an accept state.

\[\text{\scriptsize The notation } \bigcup_{r \in R} \delta(r, a) \text{ means: the union of the sets } \delta(r, a) \text{ for each possible } r \text{ in } R.\]
Now we need to consider the $\varepsilon$ arrows. To do so we set up an extra bit of notation. For any state $R$ of $M$ we define $E(R)$ to be the collection of states that can be reached from $R$ by going only along $\varepsilon$ arrows, including the members of $R$ themselves. Formally, for $R \subseteq Q$ let

$$E(R) = \{ q \mid q \text{ can be reached from } R \text{ by traveling along } 0 \text{ or more } \varepsilon \text{ arrows}\}.$$  

Then we modify the transition function of $M$ to place additional fingers on all states that can be reached by going along $\varepsilon$ arrows after every step. Replacing $\delta(r,a)$ by $E(\delta(r,a))$ achieves this effect. Thus

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

Additionally we need to modify the start state of $M$ to move the fingers initially to all possible states that can be reached from the start state of $N$ along the $\varepsilon$ arrows. Changing $q_0'$ to be $E(\{q_0\})$ achieves this effect. We have now completed the construction of the DFA $M$ that simulates the NFA $N$.

The construction of $M$ obviously works correctly. At every step in the computation of $M$ on an input, it clearly enters a state that corresponds to the subset of states that $N$ could be in at that point. Thus our proof is complete.

If the construction used in the preceding proof were more complex we would need to prove that it works as claimed. Usually such proofs proceed by induction on the number of steps of the computation. Most of the constructions that we use in this book are straightforward and so do not require such a correctness proof. An example of a more complex construction that we do prove correct appears in the proof of Theorem 1.54.

Theorem 1.39 states that every NFA can be converted into an equivalent DFA. Thus nondeterministic finite automata give an alternative way of characterizing the regular languages. We state this fact as a corollary of Theorem 1.39.

**Corollary 1.40**

A language is regular if and only if some nondeterministic finite automaton recognizes it.

One direction of the “if and only if” condition states that a language is regular if some NFA recognizes it. Theorem 1.39 shows that any NFA can be converted into an equivalent DFA. Consequently, if an NFA recognizes some language, so does some DFA, and hence the language is regular. The other direction of the “if and only if” condition states that a language is regular only if some NFA recognizes it. That is, if a language is regular, some NFA must be recognizing it. Obviously, this condition is true because a regular language has a DFA recognizing it and any DFA is also an NFA.
EXAMPLE 1.41

Let's illustrate the procedure we gave in the proof of Theorem 1.39 for converting an NFA to a DFA by using the machine $N_4$ that appears in Example 1.35. For clarity, we have relabeled the states of $N_4$ to be $\{1, 2, 3\}$. Thus in the formal description of $N_4 = (Q, \{a, b\}, \delta, 1, \{1\})$, the set of states $Q$ is $\{1, 2, 3\}$ as shown in the following figure.

To construct a DFA $D$ that is equivalent to $N_4$, we first determine $D$'s states. $N_4$ has three states, $\{1, 2, 3\}$, so we construct $D$ with eight states, one for each subset of $N_4$'s states. We label each of $D$'s states with the corresponding subset. Thus $D$'s state set is

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

![Diagram of NFA $N_4$]

FIGURE 1.42
The NFA $N_4$

Next, we determine the start and accept states of $D$. The start state is $E(\{1\})$, the set of states that are reachable from 1 by traveling along $\varepsilon$ arrows, plus 1 itself. An $\varepsilon$ arrow goes from 1 to 3, so $E(\{1\}) = \{1, 3\}$. The new accept states are those containing $N_4$'s accept state; thus $\{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$.

Finally, we determine $D$'s transition function. Each of $D$'s states goes to one place on input $a$ and one place on input $b$. We illustrate the process of determining the placement of $D$'s transition arrows with a few examples.

In $D$, state $\{2\}$ goes to $\{2, 3\}$ on input $a$, because in $N_4$, state 2 goes to both 2 and 3 on input $a$ and we can't go farther from 2 or 3 along $\varepsilon$ arrows. State $\{2\}$ goes to state $\{3\}$ on input $b$, because in $N_4$, state 2 goes only to state 3 on input $b$ and we can't go farther from 3 along $\varepsilon$ arrows.

State $\{1\}$ goes to $\emptyset$ on $a$, because no $a$ arrows exit it. It goes to $\{2\}$ on $b$. Note that the procedure in Theorem 1.39 specifies that we follow the $\varepsilon$ arrows after each input symbol is read. An alternative procedure based on following the $\varepsilon$ arrows before reading each input symbol works equally well, but that method is not illustrated in this example.

State $\{3\}$ goes to $\{1, 3\}$ on $a$, because in $N_4$, state 3 goes to 1 on $a$ and 1 in turn goes to 3 with an $\varepsilon$ arrow. State $\{3\}$ on $b$ goes to $\emptyset$. 
State \( \{1,2\} \) on \( a \) goes to \( \{2,3\} \) because 1 points at no states with \( a \) arrows and 2 points at both 2 and 3 with \( a \) arrows and neither points anywhere with \( \varepsilon \) arrows. State \( \{1,2\} \) on \( b \) goes to \( \{2,3\} \). Continuing in this way we obtain the following diagram for \( D \).

**Figure 1.43**  
A DFA \( D \) that is equivalent to the NFA \( N_4 \)

We may simplify this machine by observing that no arrows point at states \( \{1\} \) and \( \{1,2\} \), so they may be removed without affecting the performance of the machine. Doing so yields the following figure.

**Figure 1.44**  
DFA \( D \) after removing unnecessary states

**Closure Under the Regular Operations**

Now we return to the closure of the class of regular languages under the regular operations that we began in Section 1.1. Our aim is to prove that the union, concatenation, and star of regular languages are still regular. We abandoned the original attempt to do so when dealing with the concatenation operation was too complicated. The use of nondeterminism makes the proofs much easier.
First, let's consider again closure under union. Earlier we proved closure under union by simulating deterministically both machines simultaneously via a Cartesian product construction. We now give a new proof to illustrate the technique of nondeterminism. Reviewing the first proof, appearing on page 45, may be worthwhile to see how much easier and more intuitive the new proof is.

**Theorem 1.45**

The class of regular languages is closed under the union operation.

**Proof Idea** We have regular languages $A_1$ and $A_2$ and want to prove that $A_1 \cup A_2$ is regular. The idea is to take two NFAs, $N_1$ and $N_2$ for $A_1$ and $A_2$, and combine them into one new NFA, $N$.

Machine $N$ must accept its input if either $N_1$ or $N_2$ accepts this input. The new machine has a new start state that branches to the start states of the old machines with $\varepsilon$ arrows. In this way the new machine nondeterministically guesses which of the two machines accepts the input. If one of them accepts the input, $N$ will accept it, too.

We represent this construction in the following figure. On the left, we indicate the start and accept states of machines $N_1$ and $N_2$ with large circles and some additional states with small circles. On the right, we show how to combine $N_1$ and $N_2$ into $N$ by adding additional transition arrows.

**Figure 1.46**

Construction of an NFA $N$ to recognize $A_1 \cup A_2$
Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize $A_1$, and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize $A_2$.

Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.
   The states of $N$ are all the states of $N_1$ and $N_2$, with the addition of a new start state $q_0$.

2. The state $q_0$ is the start state of $N$.

3. The accept states $F = F_1 \cup F_2$.
   The accept states of $N$ are all the accept states of $N_1$ and $N_2$. That way $N$ accepts if either $N_1$ accepts or $N_2$ accepts.

4. Define $\delta$ so that for any $q \in Q$ and any $a \in \Sigma$, $$\delta(q, a) = \begin{cases} 
\delta_1(q, a) & q \in Q_1 \\
\delta_2(q, a) & q \in Q_2 \\
\{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\
\emptyset & q = q_0 \text{ and } a \neq \varepsilon.
\end{cases}$$

Now we can prove closure under concatenation. Recall that earlier, without nondeterminism, completing the proof would have been difficult.

**Theorem 1.47**

The class of regular languages is closed under the concatenation operation.

**Proof Idea** We have regular languages $A_1$ and $A_2$ and want to prove that $A_1 \circ A_2$ is regular. The idea is to take two NFAs, $N_1$ and $N_2$ for $A_1$ and $A_2$, and combine them into a new NFA $N$ as we did for the case of union, but this time in a different way, as shown in Figure 1.48.

Assign $N$'s start state to be the start state of $N_1$. The accept states of $N_1$ have additional $\varepsilon$ arrows that nondeterministically allow branching to $N_2$ whenever $N_1$ is in an accept state, signifying that it has found an initial piece of the input that constitutes a string in $A_1$. The accept states of $N$ are the accept states of $N_2$ only. Therefore it accepts when the input can be split into two parts, the first accepted by $N_1$ and the second by $N_2$. We can think of $N$ as nondeterministically guessing where to make the split.
PROOF

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize $A_1$, and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize $A_2$.

Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $A_1 \circ A_2$.

1. $Q = Q_1 \cup Q_2$.
   The states of $N$ are all the states of $N_1$ and $N_2$.

2. The state $q_1$ is the same as the start state of $N_1$.

3. The accept states $F_2$ are the same as the accept states of $N_2$.

4. Define $\delta$ so that for any $q \in Q$ and any $a \in \Sigma$, $\delta(q,a)$ is defined as follows:

$\delta(q,a) = \begin{cases} 
\delta_1(q,a) & q \in Q_1 \text{ and } a \notin F_1 \\
\delta_1(q,a) & q \in F_1 \text{ and } a \neq \epsilon \\
\delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ and } a = \epsilon \\
\delta_2(q,a) & q \in Q_2.
\end{cases}$
The class of regular languages is closed under the star operation.

**Proof Idea** We have a regular language $A_1$ and want to prove that $A_1^*$ also is regular. We take an NFA $N_1$ for $A_1$ and modify it to recognize $A_1^*$, as shown in the following figure. The resulting NFA $N$ will accept its input whenever it can be broken into several pieces and $N_1$ accepts each piece.

We can construct $N$ like $N_1$ with additional $\varepsilon$ arrows returning to the start state from the accept states. This way, when processing gets to the end of a piece that $N_1$ accepts, the machine $N$ has the option of jumping back to the start state to try to read in another piece that $N_1$ accepts. In addition we must modify $N$ so that it accepts $\varepsilon$, which always is a member of $A_1^*$. One (slightly bad) idea is simply to add the start state to the set of accept states. This approach certainly adds $\varepsilon$ to the recognized language, but it may also add other, undesired strings. Exercise 1.15 asks for an example of the failure of this idea. The way to fix it is to add a new start state, which also is an accept state, and which has an $\varepsilon$ arrow to the old start state. This solution has the desired effect of adding $\varepsilon$ to the language without adding anything else.

**Figure 1.50**
Construction of $N$ to recognize $A^*$

**Proof** Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize $A_1$.

Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $A_1^*$.

1. $Q = \{q_0\} \cup Q_1$.
   The states of $N$ are the states of $N_1$ plus a new start state.

2. The state $q_0$ is the new start state.

3. $F = \{q_0\} \cup F_1$.
   The accept states are the old accept states plus the new start state.
4. Define $\delta$ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q, a) = \begin{cases} 
\delta_1(q, a) & q \in Q_1 \text{ and } q \not\in F_1 \\
\delta_1(q, a) & q \in F_1 \text{ and } a \neq \varepsilon \\
\delta_1(q, a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon \\
\{q_1\} & q = q_0 \text{ and } a = \varepsilon \\
\emptyset & q = q_0 \text{ and } a \neq \varepsilon.
\end{cases}$$

1.3

REGULAR EXPRESSIONS

In arithmetic, we can use the operations $+$ and $\times$ to build up expressions such as

$$(5 + 3) \times 4.$$ 

Similarly, we can use the regular operations to build up expressions describing languages, which are called regular expressions. An example is:

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

The value of the arithmetic expression is the number 32. The value of a regular expression is a language. In this case the value is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s. We get this result by dissecting the expression into its parts. First, the symbols 0 and 1 are shorthand for the sets $\{0\}$ and $\{1\}$. So $(0 \cup 1)$ means $\{0\} \cup \{1\}$. The value of this part is the language $\{0, 1\}$. The part $0^*$ means $\{0\}^*$, and its value is the language consisting of all strings containing any number of 0s. Second, like the $\times$ symbol in algebra, the concatenation symbol $\circ$ often is implicit in regular expressions. Thus $(0 \cup 1)^* \circ 0^*$ actually is shorthand for $(0 \cup 1) \circ 0^*$. The concatenation attaches the strings from the two parts to obtain the value of the entire expression.

Regular expressions have an important role in computer science applications. In applications involving text, users may want to search for strings that satisfy certain patterns. Regular expressions provide a powerful method for describing such patterns. Utilities such as AWK and GREP in UNIX, modern programming languages such as PERL, and text editors all provide mechanisms for the description of patterns by using regular expressions.
Another example of a regular expression is

\[(0 \cup 1)^*\]

It starts with the language \((0 \cup 1)\) and applies the * operation. The value of this expression is the language consisting of all possible strings of 0s and 1s. If \(\Sigma = \{0, 1\}\), we can write \(\Sigma\) as shorthand for the regular expression \((0 \cup 1)\). More generally, if \(\Sigma\) is any alphabet, the regular expression \(\Sigma\) describes the language consisting of all strings of length 1 over this alphabet, and \(\Sigma^*\) describes the language consisting of all strings over that alphabet. Similarly \(\Sigma^* 1\) is the language that contains all strings that end in a 1. The language \((0\Sigma^*) \cup (\Sigma^* 1)\) consists of all strings that either start with a 0 or end with a 1.

In arithmetic, we say that \(\times\) has precedence over + to mean that, when there is a choice, we do the \(\times\) operation first. Thus in \(2 + 3 \times 4\) the \(3 \times 4\) is done before the addition. To have the addition done first we must add parentheses to obtain \((2 + 3) \times 4\). In regular expressions, the star operation is done first, followed by concatenation, and finally union, unless parentheses are used to change the usual order.

**Formal Definition of a Regular Expression**

**Definition 1.52**

Say that \(R\) is a *regular expression* if \(R\) is

1. \(a\) for some \(a\) in the alphabet \(\Sigma\),
2. \(\varepsilon\),
3. \(\emptyset\),
4. \((R_1 \cup R_2)\), where \(R_1\) and \(R_2\) are regular expressions,
5. \((R_1 \circ R_2)\), where \(R_1\) and \(R_2\) are regular expressions, or
6. \((R_1^*)\), where \(R_1\) is a regular expression.

In items 1 and 2, the regular expressions \(a\) and \(\varepsilon\) represent the languages \(\{a\}\) and \(\{\varepsilon\}\), respectively. In item 3, the regular expression \(\emptyset\) represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages \(R_1\) and \(R_2\), or the star of the language \(R_1\), respectively.

Don’t confuse the regular expressions \(\varepsilon\) and \(\emptyset\). The expression \(\varepsilon\) represents the language containing a single string—namely, the empty string—whereas \(\emptyset\) represents the language that doesn’t contain any strings.
Seemingly, we are in danger of defining the notion of regular expression in terms of itself. If true, we would have a circular definition, which would be invalid. However, $R_1$ and $R_2$ always are smaller than $R$. Thus we actually are defining regular expressions in terms of smaller regular expressions and thereby avoiding circularity. A definition of this type is called an inductive definition.

Parentheses in an expression may be omitted. If they are, evaluation is done in the precedence order: star, then concatenation, then union.

For convenience, we let $R^*$ be shorthand for $RR^*$. In other words, whereas $R^*$ has all strings that are 0 or more concatenations of strings from $R$, the language $R^*$ has all strings that that are 1 or more concatenations of strings from $R$. So $R^* \cup \varepsilon = R^*$. In addition, we let $R^k$ be shorthand for the concatenation of $k$ $R$’s with each other.

When we want to distinguish between a regular expression $R$ and the language that it describes, we write $L(R)$ to be the language of $R$.

**Example 1.53**

In the following instances we assume that the alphabet $\Sigma$ is \{0,1\}.

1. $0^*10^* = \{w | w \text{ contains a single 1}\}$.
2. $\Sigma^*1\Sigma^* = \{w | w \text{ has at least one 1}\}$.
3. $\Sigma^*001\Sigma^* = \{w | w \text{ contains the string 001 as a substring}\}$.
4. $1^*(01^*)^* = \{w | \text{every 0 in } w \text{ is followed by at least one 1}\}$.
5. $(\Sigma\Sigma)^* = \{w | w \text{ is a string of even length}\}$.\(^5\)
6. $(\Sigma\Sigma\Sigma)^* = \{w | \text{the length of } w \text{ is a multiple of three}\}$.
7. $01 \cup 10 = \{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w | w \text{ starts and ends with the same symbol}\}$.
9. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.
   The expression $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$, so the concatenation operation adds either 0 or $\varepsilon$ before every string in $1^*$.
10. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
11. $1^*\emptyset = \emptyset$.
   Concatenating the empty set to any set yields the empty set.
12. $\emptyset^* = \{\varepsilon\}$.
   The star operation puts together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, giving only the empty string.

\(^5\)The length of a string is the number of symbols that it contains.
If we let \( R \) be any regular expression, we have the following identities. They are good tests of whether you understand the definition.

\[
R \cup \emptyset = R.
\]
Adding the empty language to any other language will not change it.

\[
R \circ \varepsilon = R.
\]
Joining the empty string to any string will not change it.

However, exchanging \( \emptyset \) and \( \varepsilon \) in the preceding identities may cause the equalities to fail.

\[
R \cup \varepsilon \text{ may not equal } R.
\]
For example, if \( R = 0 \), then \( L(R) = \{0\} \) but \( L(R \cup \varepsilon) = \{0, \varepsilon\} \).

\[
R \circ \emptyset \text{ may not equal } R.
\]
For example, if \( R = 0 \), then \( L(R) = \{0\} \) but \( L(R \circ \emptyset) = \emptyset \).

Regular expressions are useful tools in the design of compilers for programming languages. Elemental objects in a programming language, called tokens, such as the variable names and constants, may be described with regular expressions. For example, a numerical constant that may include a fractional part and/or a sign may be described as a member of the language

\[
(\pm \cup - \cup \varepsilon) \,(D^+ \cup D^+ \cdot D^+ \cup D^+ \cdot D^+)
\]
where \( D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \) is the alphabet of decimal digits. Examples of generated strings are: 72, 3.14159, +7., and -.01.

Once the syntax of the tokens of the programming language have been described with regular expressions, automatic systems can generate the lexical analyzer, the part of a compiler that initially processes the input program.

**EQUIVALENCE WITH FINITE AUTOMATA**

Regular expressions and finite automata are equivalent in their descriptive power. This fact is surprising because finite automata and regular expressions superficially appear to be rather different. However, any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa. Recall that a regular language is one that is recognized by some finite automaton.

**THEOREM 1.54**

A language is regular if and only if some regular expression describes it.

This theorem has two directions. We state and prove each direction as a separate lemma.
LEMMA 1.55

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

PROOF IDEA Say that we have a regular expression $R$ describing some language $A$. We show how to convert $R$ into an NFA recognizing $A$. By Corollary 1.40, if an NFA recognizes $A$ then $A$ is regular.

PROOF Let's convert $R$ into an NFA $N$. We consider the six cases in the formal definition of regular expressions.

1. $R = a$ for some $a$ in $\Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.

   ![Diagram 1](chart1.png)

   Note that this machine fits the definition of an NFA but not that of a DFA because it has some states with no exiting arrow for each possible input symbol. Of course, we could have presented an equivalent DFA here but an NFA is all we need for now, and it is easier to describe.

   Formally, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where we describe $\delta$ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.

2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$, and the following NFA recognizes $L(R)$.

   ![Diagram 2](chart2.png)

   Formally, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, where $\delta(r, b) = \emptyset$ for any $r$ and $b$.

3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$.

   ![Diagram 3](chart3.png)

   Formally, $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, where $\delta(r, b) = \emptyset$ for any $r$ and $b$.

4. $R = R_1 \cup R_2$.

5. $R = R_1 \circ R_2$.

6. $R = R_1^*$.

For the last three cases we use the constructions given in the proofs that the class of regular languages is closed under the regular operations. In other words, we construct the NFA for $R$ from the NFAs for $R_1$ and $R_2$ (or just $R_1$ in case 6) and the appropriate closure construction.
That ends the first part of the proof of Theorem 1.54, giving the easier direction of the if and only if condition. Before going on to the other direction, let's consider some examples whereby we use this procedure to convert a regular expression to an NFA.

**Example 1.56**

We convert the regular expression \((ab \cup a)^*\) to an NFA in a sequence of stages. We build up from the smallest subexpressions to larger subexpressions until we have an NFA for the original expression, as shown in the following diagram. Note that this procedure generally doesn't give the NFA with the fewest states. In this example, the procedure gives an NFA with eight states, but the smallest equivalent NFA has only two states. Can you find it?

![Diagram](image)

**Figure 1.57**
Building an NFA from the regular expression \((ab \cup a)^*\)
**Example 1.58**

In Figure 1.59, we convert the regular expression \((a \cup b)^*aba\) to an NFA. A few of the minor steps are not shown.

![Diagram of NFA for \((a \cup b)^*aba\)](image)

**Figure 1.59**
Building an NFA from the regular expression \((a \cup b)^*aba\)

Now let's turn to the other direction of the proof of Theorem 1.54.

**Lemma 1.60**

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

**Proof Idea** We need to show that, if a language \(A\) is regular, a regular expression describes it. Because \(A\) is regular, it is accepted by a DFA. We describe a procedure for converting DFAs into equivalent regular expressions.
We break this procedure into two parts, using a new type of finite automaton called a generalized nondeterministic finite automaton, GNFA. First we show how to convert DFAs into GNFAs, and then GNFAs into regular expressions.

Generalized nondeterministic finite automata are simply nondeterministic finite automata wherein the transition arrows may have any regular expressions as labels, instead of only members of the alphabet or $\varepsilon$. The GNFA reads blocks of symbols from the input, not necessarily just one symbol at a time as in an ordinary NFA. The GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string described by the regular expression on that arrow. A GNFA is nondeterministic and so may have several different ways to process the same input string. It accepts its input if its processing can cause the GNFA to be in an accept state at the end of the input. The following figure presents an example of a GNFA.

![Diagram of a GNFA](image)

**FIGURE 1.61**
A generalized nondeterministic finite automaton

For convenience we require that GNFAs always have a special form that meets the following conditions.

- The start state has transition arrows going to every other state but no arrows coming in from any other state.
- There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
- Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.
1.3 REGULAR EXPRESSIONS

We can easily convert a DFA into a GNFA in the special form. We simply add a new start state with an $\varepsilon$ arrow to the old start state and a new accept state with $\varepsilon$ arrows from the old accept states. If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels. Finally, we add arrows labeled $\emptyset$ between states that had no arrows. This last step won’t change the language recognized because a transition labeled with $\emptyset$ can never be used. From here on we assume that all GNFAs are in the special form.

Now we show how to convert a GNFA into a regular expression. Say that the GNFA has $k$ states. Then, because a GNFA must have a start and an accept state and they must be different from each other, we know that $k \geq 2$. If $k > 2$, we construct an equivalent GNFA with $k - 1$ states. This step can be repeated on the new GNFA until it is reduced to two states. If $k = 2$, the GNFA has a single arrow that goes from the start state to the accept state. The label of this arrow is the equivalent regular expression. For example, the stages in converting a DFA with three states to an equivalent regular expression are shown in the following figure.

![Diagram](image)

**FIGURE 1.62**
Typical stages in converting a DFA to a regular expression

The crucial step is in constructing an equivalent GNFA with one fewer state when $k > 2$. We do so by selecting a state, ripping it out of the machine, and repairing the remainder so that the same language is still recognized. Any state will do, provided that it is not the start or accept state. We are guaranteed that such a state will exist because $k > 2$. Let’s call the removed state $q_{\text{rip}}$.

After removing $q_{\text{rip}}$ we repair the machine by altering the regular expressions that label each of the remaining arrows. The new labels compensate for the absence of $q_{\text{rip}}$ by adding back the lost computations. The new label going from a state $q_i$ to a state $q_j$ is a regular expression that describes all strings that would
take the machine from $q_i$ to $q_j$ either directly or via $q_{\text{rip}}$. We illustrate this approach in Figure 1.63.

![Diagram showing transition from $q_i$ to $q_j$](image)

**Figure 1.63**
Constructing an equivalent GNFA with one fewer state

In the old machine if $q_i$ goes to $q_{\text{rip}}$ with an arrow labeled $R_1$, $q_{\text{rip}}$ goes to itself with an arrow labeled $R_2$, $q_{\text{rip}}$ goes to $q_j$ with an arrow labeled $R_3$, and $q_i$ goes to $q_j$ with an arrow labeled $R_4$, then in the new machine the arrow from $q_i$ to $q_j$ gets the label

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

We make this change for each arrow going from any state $q_i$ to any state $q_j$, including the case where $q_i = q_j$. The new machine recognizes the original language.

**Proof** Let’s now carry out this idea formally. First, to facilitate the proof, we formally define the new type of automaton introduced. A GNFA is similar to a nondeterministic finite automaton except for the transition function, which has the form

$$\delta: (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \rightarrow \mathcal{R}.$$ 

The symbol $\mathcal{R}$ is the collection of all regular expressions over the alphabet $\Sigma$, and $q_{\text{start}}$ and $q_{\text{accept}}$ are the start and accept states. If $\delta(q_i, q_j) = R$, the arrow from state $q_i$ to state $q_j$ has the regular expression $R$ as its label. The domain of the transition function is $(Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\})$ because an arrow connects every state to every other state, except that no arrows are coming from $q_{\text{accept}}$ or going to $q_{\text{start}}$. 
A generalized nondeterministic finite automaton is a 5-tuple, $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where

1. $Q$ is the finite set of states,
2. $\Sigma$ is the input alphabet,
3. $\delta : (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \rightarrow \mathcal{R}$ is the transition function,
4. $q_{\text{start}}$ is the start state, and
5. $q_{\text{accept}}$ is the accept state.

A GNFA accepts a string $w$ in $\Sigma^*$ if $w = w_1 w_2 \ldots w_k$, where each $w_i$ is in $\Sigma^*$ and a sequence of states $q_0, q_1, \ldots, q_k$ exists such that

1. $q_0 = q_{\text{start}}$ is the start state,
2. $q_k = q_{\text{accept}}$ is the accept state, and
3. for each $i$, we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$; in other words, $R_i$ is the expression on the arrow from $q_{i-1}$ to $q_i$.

Returning to the proof of Lemma 1.60, we let $M$ be the DFA for language $A$. Then we convert $M$ to a GNFA $G$ by adding a new start state and a new accept state and additional transition arrows as necessary. We use the procedure $\text{CONVERT}(G)$, which takes a GNFA and returns an equivalent regular expression. This procedure uses recursion, which means that it calls itself. An infinite loop is avoided because the procedure calls itself only to process a GNFA that has one fewer state. The case where the GNFA has two states is handled without recursion.

$\text{CONVERT}(G)$:

1. Let $k$ be the number of states of $G$.
2. If $k = 2$, then $G$ must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression $R$.
   Return the expression $R$.
3. If $k > 2$, we select any state $q_{\text{trip}} \in Q$ different from $q_{\text{start}}$ and $q_{\text{accept}}$ and let $G'$ be the GNFA $(Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$, where
   $$ Q' = Q - \{q_{\text{trip}}\}, $$
   and for any $q_i \in Q' - \{q_{\text{accept}}\}$ and any $q_j \in Q' - \{q_{\text{start}}\}$ let
   $$ \delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4), $$
   for $R_1 = \delta(q_i, q_{\text{trip}})$, $R_2 = \delta(q_{\text{trip}}, q_{\text{trip}})$, $R_3 = \delta(q_{\text{trip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$.
4. Compute $\text{CONVERT}(G')$ and return this value.
Next we prove that CONVERT returns a correct value.

**Claim 1.65**

For any GNFA $G$, $\text{CONVERT}(G)$ is equivalent to $G$.

We prove this claim by induction on $k$, the number of states of the GNFA.

**Basis:** Prove the claim true for $k = 2$ states. If $G$ has only two states, it can have only a single arrow, which goes from the start state to the accept state. The regular expression label on this arrow describes all the strings that allow $G$ to get to the accept state. Hence this expression is equivalent to $G$.

**Induction step:** Assume that the claim is true for $k - 1$ states and use this assumption to prove that the claim is true for $k$ states. First we show that $G$ and $G'$ recognize the same language. Suppose that $G$ accepts an input $w$. Then in an accepting branch of the computation $G$ enters a sequence of states:

$$q_{\text{start}}, q_1, q_2, q_3, \ldots, q_{\text{accept}}.$$

If none of them is the removed state $q_{\text{trip}}$, clearly $G'$ also accepts $w$. The reason is that each of the new regular expressions labeling the arrows of $G'$ contains the old regular expression as part of a union.

If $q_{\text{trip}}$ does appear, removing each run of consecutive $q_{\text{trip}}$ states forms an accepting computation for $G'$. The states $q_i$ and $q_j$ bracketing a run have a new regular expression on the arrow between them that describes all strings taking $q_i$ to $q_j$ via $q_{\text{trip}}$ on $G$. So $G'$ accepts $w$.

Conversely, suppose that $G'$ accepts an input $w$. As each arrow between any two states $q_i$ and $q_j$ in $G'$ describes the collection of strings taking $q_i$ to $q_j$ in $G$, either directly or via $q_{\text{trip}}$, $G$ must also accept $w$. Thus $G$ and $G'$ are equivalent.

The induction hypothesis states that when the algorithm calls itself recursively on input $G'$, the result is a regular expression that is equivalent to $G'$ because $G'$ has $k - 1$ states. Hence this regular expression also is equivalent to $G$, and the algorithm is proved correct.

This concludes the proof of Claim 1.65, Lemma 1.60, and Theorem 1.54.

**Example 1.66**

In this example we use the preceding algorithm to convert a DFA into a regular expression. We begin with the two-state DFA in Figure 1.67(a).

In Figure 1.67(b) we make a four-state GNFA by adding a new start state and a new accept state, called $s$ and $a$ instead of $q_{\text{start}}$ and $q_{\text{accept}}$ so that we can draw them conveniently. To avoid cluttering up the figure, we do not draw the arrows
labeled $\emptyset$, even though they are present. Note that we replace the label $a, b$ on the self-loop at state 2 on the DFA with the label $a \cup b$ at the corresponding point on the GNFA. We do so because the DFA's label represents two transitions, one for $a$ and the other for $b$, whereas the GNFA may have only a single transition going from 2 to itself.

In Figure 1.67(c) we remove state 2, and update the remaining arrow labels. In this case the only label that changes is the one from 1 to $a$. In part (b) it was $\emptyset$, but in part (c) it is $b(a \cup b)^*$. We obtain this result by following step 3 of the CONVERT procedure. State $q_i$ is state 1, state $q_j$ is $a$, and $q_{flip}$ is 2, so $R_1 = b$, $R_2 = a \cup b$, $R_3 = \epsilon$, and $R_4 = \emptyset$. Therefore the new label on the arrow from 1 to $a$ is $(b)(a \cup b)^*(\epsilon) \cup \emptyset$. We simplify this regular expression to $b(a \cup b)^*$.

In Figure 1.67(d) we remove state 1 from part (c) and follow the same procedure. Because only the start and accept states remain, the label on the arrow joining them is the regular expression that is equivalent to the original DFA.

\begin{figure}[h]
\centering
\begin{subfigure}{0.4\textwidth}
\centering
\begin{tikzpicture}
  \node[state, initial] (1) at (0,0) {1} ;
  \node[state, below of=1] (2) {2} ;
  \draw[->] (1) edge [loop above] node {$a$} () ;
  \draw[->] (1) edge [loop below] node {$b$} () ;
  \draw[->] (2) edge [loop above] node {$a, b$} () ;
  \draw[->] (1) edge node {$b$} (2) ;
\end{tikzpicture}
\caption{(a)}
\end{subfigure}
\hfill
\begin{subfigure}{0.4\textwidth}
\centering
\begin{tikzpicture}
  \node[state, initial] (1) at (0,0) {1} ;
  \node[state, below of=1] (2) {2} ;
  \draw[->] (1) edge [loop above] node {$a$} () ;
  \draw[->] (1) edge [loop below] node {$b$} () ;
  \draw[->] (2) edge [loop above] node {$a \cup b$} () ;
  \draw[->] (1) edge node {$\epsilon$} (2) ;
\end{tikzpicture}
\caption{(b)}
\end{subfigure}
\hfill
\begin{subfigure}{0.4\textwidth}
\centering
\begin{tikzpicture}
  \node[state, initial] (1) at (0,0) {1} ;
  \node[state, below of=1] (a) {a} ;
  \draw[->] (1) edge [loop above] node {$a$} () ;
  \draw[->] (1) edge [loop below] node {$b(a \cup b)^*$} () ;
  \draw[->] (a) edge node {$\epsilon$} (1) ;
\end{tikzpicture}
\caption{(c)}
\end{subfigure}
\hfill
\begin{subfigure}{0.4\textwidth}
\centering
\begin{tikzpicture}
  \node[state, initial] (1) at (0,0) {1} ;
  \node[state, below of=1] (a) {a} ;
  \draw[->] (1) edge [loop above] node {$a$} () ;
  \draw[->] (1) edge [loop below] node {$a^*b(a \cup b)^*$} () ;
  \draw[->] (a) edge node {$\epsilon$} (1) ;
\end{tikzpicture}
\caption{(d)}
\end{subfigure}
\caption{Converting a two-state DFA to an equivalent regular expression}
\end{figure}
EXAMPLE 1.68

In this example we begin with a three-state DFA. The steps in the conversion are shown in the following figure.

(a) ![Diagram of a three-state DFA](image1)

(b) ![Diagram of a three-state DFA](image2)

(c) ![Diagram of a three-state DFA](image3)

(d) ![Diagram of a three-state DFA](image4)

(e) ![Diagram of a three-state DFA](image5)

\[(a(aa \cup b)^* ab \cup b)((ba \cup a)(aa \cup b)^* ab \cup bb)^* ((ba \cup a)(aa \cup b)^* \varepsilon) \cup a(aa \cup b)^*\]

**Figure 1.69**

Converting a three-state DFA to an equivalent regular expression
1.4 NONREGULAR LANGUAGES

To understand the power of finite automata you must also understand their limitations. In this section we show how to prove that certain languages cannot be recognized by any finite automaton.

Let’s take the language \( B = \{0^n1^n \mid n \geq 0\} \). If we attempt to find a DFA that recognizes \( B \), we discover that the machine seems to need to remember how many 0s have been seen so far as it reads the input. Because the number of 0s isn’t limited, the machine will have to keep track of an unlimited number of possibilities. But it cannot do so with any finite number of states.

Next, we present a method for proving that languages such as \( B \) are not regular. Doesn’t the argument already given prove nonregularity because the number of 0s is unlimited? It does not. Just because the language appears to require unbounded memory doesn’t mean that it is necessarily so. It does happen to be true for the language \( B \), but other languages seem to require an unlimited number of possibilities, yet actually they are regular. For example, consider two languages over the alphabet \( \Sigma = \{0,1\} \):

\[
C = \{w \mid w \text{ has an equal number of 0s and 1s}\}, \text{ and}
\]

\[
D = \{w \mid w \text{ has an equal number of occurrences of 01 and 10 as substrings}\}.
\]

At first glance a recognizing machine appears to need to count in each case, and therefore neither language appears to be regular. As expected, \( C \) is not regular, but surprisingly \( D \) is regular.\(^6\) Thus our intuition can sometimes lead us astray, which is why we need mathematical proofs for certainty. In this section we show how to prove that certain languages are not regular.

THE PUMPING LEMMA FOR REGULAR LANGUAGES

Our technique for proving nonregularity stems from a theorem about regular languages, traditionally called the pumping lemma. This theorem states that all regular languages have a special property. If we can show that a language does not have this property, we are guaranteed that it is not regular. The property states that all strings in the language can be “pumped” if they are at least as long as a certain special value, called the pumping length. That means each such string contains a section that can be repeated any number of times with the resulting string remaining in the language.

---

\(^6\)See Problem 1.48.
THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into three pieces, \( s = xyz \), satisfying the following conditions:

1. for each \( i \geq 0 \), \( xy^iz \in A \),
2. \( |y| > 0 \), and
3. \( |xy| \leq p \).

Recall the notation where \( |s| \) represents the length of string s, \( y^i \) means that i copies of y are concatenated together, and \( y^0 \) equals \( \varepsilon \).

When s is divided into \( xyz \), either x or z may be \( \varepsilon \), but condition 2 says that \( y \neq \varepsilon \). Observe that without condition 2 the theorem would be trivially true. Condition 3 states that the pieces x and y together have length at most p. It is an extra technical condition that we occasionally find useful when proving certain languages to be nonregular. See Example 1.74 for an application of condition 3.

PROOF IDEA Let \( M = (Q, \Sigma, \delta, q_1, F) \) be a DFA that recognizes A. We assign the pumping length \( p \) to be the number of states of \( M \). We show that any string \( s \) in A of length at least \( p \) may be broken into the three pieces \( xyz \) satisfying our three conditions. What if no strings in A are of length at least \( p \)? Then our task is even easier because the theorem becomes vacuously true: Obviously the three conditions hold for all strings of length at least \( p \) if there aren’t any such strings.

If \( s \) in A has length at least \( p \), consider the sequence of states that \( M \) goes through when computing with input \( s \). It starts with \( q_1 \) the start state, then goes to, say, \( q_3 \), then, say, \( q_{20} \), then \( q_9 \), and so on, until it reaches the end of \( s \) in state \( q_{13} \). With \( s \) in A, we know that \( M \) accepts \( s \), so \( q_{13} \) is an accept state.

If we let \( n \) be the length of \( s \), the sequence of states \( q_1, q_3, q_{20}, q_9, \ldots, q_{13} \) has length \( n + 1 \). Because \( n \) is at least \( p \), we know that \( n + 1 \) is greater than \( p \), the number of states of \( M \). Therefore the sequence must contain a repeated state. This result is an example of the pigeonhole principle, a fancy name for the rather obvious fact that if \( p \) pigeons are placed into fewer than \( p \) holes, some hole has to have more than one pigeon in it.

The following figure shows the string \( s \) and the sequence of states that \( M \) goes through when processing \( s \). State \( q_9 \) is the one that repeats.

\[
\begin{align*}
S &= s_1 s_2 s_3 s_4 s_5 s_6 \ldots s_n \\
q_1 &\uparrow q_3 \uparrow q_{20} \uparrow q_9 \uparrow q_6 \uparrow q_{35} \uparrow q_{13}
\end{align*}
\]

FIGURE 1.71
Example showing state \( q_9 \) repeating when \( M \) reads \( s \)

We now divide \( s \) into the three pieces \( x, y, \) and \( z \). Piece \( x \) is the part of \( s \)
appearing before \( q_9 \), piece \( y \) is the part between the two appearances of \( q_9 \), and piece \( z \) is the remaining part of \( s \), coming after the second occurrence of \( q_9 \). So \( x \) takes \( M \) from the state \( q_1 \) to \( q_9 \), \( y \) takes \( M \) from \( q_9 \) back to \( q_9 \) and \( z \) takes \( M \) from \( q_9 \) to the accept state \( q_{13} \), as shown in the following figure.

![Diagram](image)

**FIGURE 1.72**
Example showing how the strings \( x \), \( y \), and \( z \) affect \( M \)

Let’s see why this division of \( s \) satisfies the three conditions. Suppose that we run \( M \) on input \( xyyz \). We know that \( x \) takes \( M \) from \( q_1 \) to \( q_9 \), and then the first \( y \) takes it from \( q_9 \) back to \( q_9 \), as does the second \( y \), and then \( z \) takes it to \( q_{13} \). With \( q_{13} \) being an accept state, \( M \) accepts input \( xyyz \). Similarly, it will accept \( xy^i z \) for any \( i > 0 \). For the case \( i = 0 \), \( xy^0 z = xz \), which is accepted for similar reasons. That establishes condition 1.

Checking condition 2, we see that \( |y| > 0 \), as it was the part of \( s \) that occurred between two different occurrences of state \( q_9 \).

In order to get condition 3, we make sure that \( q_9 \) is the first repetition in the sequence. By the pigeonhole principle, the first \( p + 1 \) states in the sequence must contain a repetition. Therefore \( |xy| \leq p \).

**PROOF**
Let \( M = (Q, \Sigma, \delta, q_1, F) \) be a DFA recognizing \( A \) and \( p \) be the number of states of \( M \).

Let \( s = s_1 s_2 \cdots s_n \) be a string in \( A \) of length \( n \), where \( n \geq p \). Let \( r_1, \ldots, r_{n+1} \) be the sequence of states that \( M \) enters while processing \( s \), so \( r_{i+1} = \delta(r_i, s_i) \) for \( 1 \leq i \leq n \). This sequence has length \( n + 1 \), which is at least \( p + 1 \). Among the first \( p + 1 \) elements in the sequence, two must be the same state, by the pigeonhole principle. We call the first of these \( r_j \) and the second \( r_l \). Because \( r_l \) occurs among the first \( p + 1 \) places in a sequence starting at \( r_1 \), we have \( l \leq p + 1 \). Now let \( x = s_1 \cdots s_{j-1}, y = s_j \cdots s_{l-1}, \) and \( z = s_l \cdots s_n \).

As \( x \) takes \( M \) from \( r_1 \) to \( r_j \), \( y \) takes \( M \) from \( r_j \) to \( r_j \), and \( z \) takes \( M \) from \( r_j \) to \( r_{n+1} \), which is an accept state, \( M \) must accept \( xy^i z \) for \( i \geq 0 \). We know that \( j \neq l \), so \( |y| > 0 \); and \( l \leq p + 1 \), so \( |xy| \leq p \). Thus we have satisfied all conditions of the pumping lemma.
To use the pumping lemma to prove that a language $B$ is not regular, first assume that $B$ is regular in order to obtain a contradiction. Then use the pumping lemma to guarantee the existence of a pumping length $p$ such that all strings of length $p$ or greater in $B$ can be pumped. Next, find a string $s$ in $B$ that has length $p$ or greater but that cannot be pumped. Finally, demonstrate that $s$ cannot be pumped by considering all ways of dividing $s$ into $x$, $y$, and $z$ (taking condition 3 of the pumping lemma into account if convenient) and, for each such division, finding a value $i$ where $xy^iz \notin B$. This final step often involves grouping the various ways of dividing $s$ into several cases and analyzing them individually. The existence of $s$ contradicts the pumping lemma if $B$ were regular. Hence $B$ cannot be regular.

Finding $s$ sometimes takes a bit of creative thinking. You may need to hunt through several candidates for $s$ before you discover one that works. Try members of $B$ that seem to exhibit the "essence" of $B$'s nonregularity. We further discuss the task of finding $s$ in some of the following examples.

**Example 1.73**

Let $B$ be the language $\{0^n1^n | n \geq 0\}$. We use the pumping lemma to prove that $B$ is not regular. The proof is by contradiction.

Assume to the contrary that $B$ is regular. Let $p$ be the pumping length given by the pumping lemma. Choose $s$ to be the string $0^p1^p$. Because $s$ is a member of $B$ and $s$ has length more than $p$, the pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^iz$ is in $B$. We consider three cases to show that this result is impossible.

1. The string $y$ consists only of 0s. In this case the string $xyyz$ has more 0s than 1s and so is not a member of $B$, violating condition 1 of the pumping lemma. This case is a contradiction.

2. The string $y$ consists only of 1s. This case also gives a contradiction.

3. The string $y$ consists of both 0s and 1s. In this case the string $xyyz$ may have the same number of 0s and 1s, but they will be out of order with some 1s before 0s. Hence it is not a member of $B$, which is a contradiction.

Thus a contradiction is unavoidable if we make the assumption that $B$ is regular, so $B$ is not regular. Note that we can simplify this argument by applying condition 3 of the pumping lemma to eliminate cases 2 and 3.

In this example, finding the string $s$ was easy, because any string in $B$ of length $p$ or more would work. In the next two examples some choices for $s$ do not work, so additional care is required.

**Example 1.74**

Let $C = \{w | w$ has an equal number of 0s and 1s\}. We use the pumping lemma to prove that $C$ is not regular. The proof is by contradiction.
Assume to the contrary that $C$ is regular. Let $p$ be the pumping length given by the pumping lemma. As in Example 1.73, let $s$ be the string $0^p1^p$. With $s$ being a member of $C$ and having length more than $p$, the pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^iz$ is in $C$. We would like to show that this outcome is impossible. But wait, it is possible! If we let $x$ and $z$ be the empty string and $y$ be the string $0^p1^p$, then $xy^iz$ always has an equal number of 0s and 1s and hence is in $C$. So it seems that $s$ can be pumped.

Here condition 3 in the pumping lemma is useful. It stipulates that when pumping $s$ it must be divided so that $|xy| \leq p$. That restriction on the way that $s$ may be divided makes it easier to show that the string $s = 0^p1^p$ we selected cannot be pumped. If $|xy| \leq p$, then $y$ must consist only of 0s, so $xyyz \notin C$. Therefore $s$ cannot be pumped. That gives us the desired contradiction.\footnote{We could have used condition 3 in Example 1.73, as well, to simplify its proof.}

Selecting the string $s$ in this example required more care than in Example 1.73. If we had chosen $s = (01)^p$ instead, we would have run into trouble because we need a string that cannot be pumped and that string can be pumped, even taking condition 3 into account. Can you see how to pump it? One way to do so sets $x = \varepsilon$, $y = 01$, and $z = (01)^{p-1}$. Then $xy^iz \in C$ for every value of $i$. If you fail on your first attempt to find a string that cannot be pumped, don’t despair. Try another one!

An alternative method of proving that $C$ is nonregular follows from our knowledge that $B$ is nonregular. If $C$ were regular, $C \cap 0^*1^*$ also would be regular. The reasons are that the language $0^*1^*$ is regular and that the class of regular languages is closed under intersection, which we proved in footnote 3 (page 46). But $C \cap 0^*1^*$ equals $B$, and we know that $B$ is nonregular from Example 1.73.

**EXAMPLE 1.75**

Let $F = \{ww | w \in \{0,1\}^*\}$. We show that $F$ is nonregular, using the pumping lemma.

Assume to the contrary that $F$ is regular. Let $p$ be the pumping length given by the pumping lemma. Let $s$ be the string $0^p1^p$. Because $s$ is a member of $F$ and $s$ has length more than $p$, the pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, satisfying the three conditions of the lemma. We show that this outcome is impossible.

Condition 3 is once again crucial, because without it we could pump $s$ if we let $x$ and $z$ be the empty string. With condition 3 the proof follows because $y$ must consist only of 0s, so $xyyz \notin F$.

Observe that we chose $s = 0^p1^p$ to be a string that exhibits the “essence” of the nonregularity of $F$, as opposed to, say, the string $0^p0^p$. Even though $0^p0^p$ is a member of $F$, it fails to demonstrate a contradiction because it can be pumped.
EXAMPLE 1.76

Here we demonstrate a nonregular unary language. Let $D = \{1^{n^2} | n \geq 0\}$. In other words, $D$ contains all strings of 1s whose length is a perfect square. We use the pumping lemma to prove that $D$ is not regular. The proof is by contradiction.

Assume to the contrary that $D$ is regular. Let $p$ be the pumping length given by the pumping lemma. Let $s$ be the string $1^{p^2}$. Because $s$ is a member of $D$ and $s$ has length at least $p$, the pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^iz$ is in $D$. As in the preceding examples, we show that this outcome is impossible. Doing so in this case requires a little thought about the sequence of perfect squares:

\[ 0, 1, 4, 9, 16, 25, 36, 49, \ldots \]

Note the growing gap between successive members of this sequence. Large members of this sequence cannot be near each other.

Now consider the two strings $xyz$ and $xy^2z$. These strings differ from each other by a single repetition of $y$, and consequently their lengths differ by the length of $y$. By condition 3 of the pumping lemma, $|xy| \leq p$ and thus $|y| \leq p$. We have $|xyz| = p^2$ and so $|xy^2z| \leq p^2 + p$. But $p^2 + p < p^2 + 2p + 1 = (p + 1)^2$. Moreover, condition 2 implies that $y$ is not the empty string and so $|xy^2z| > p^2$. Therefore the length of $xy^2z$ lies strictly between the consecutive perfect squares $p^2$ and $(p + 1)^2$. Hence this length cannot be a perfect square itself. So we arrive at the contradiction $xy^2z \notin D$ and conclude that $D$ is not regular.

EXAMPLE 1.77

Sometimes “pumping down” is useful when we apply the pumping lemma. We use the pumping lemma to show that $E = \{0^i1^j | i > j\}$ is not regular. The proof is by contradiction.

Assume that $E$ is regular. Let $p$ be the pumping length for $E$ given by the pumping lemma. Let $s = 0^{p+1}1^p$. Then $s$ can be split into $xyz$, satisfying the conditions of the pumping lemma. By condition 3, $y$ consists only of 0s. Let’s examine the string $xyyz$ to see whether it can be in $E$. Adding an extra copy of $y$ increases the number of 0s. But, $E$ contains all strings in $0^*1^*$ that have more 0s than 1s, so increasing the number of 0s will still give a string in $E$. No contradiction occurs. We need to try something else.

The pumping lemma states that $xy^iz \in E$ even when $i = 0$, so let’s consider the string $xy^0z = xz$. Removing string $y$ decreases the number of 0s in $s$. Recall that $s$ has just one more 0 than 1. Therefore $xz$ cannot have more 0s than 1s, so it cannot be a member of $E$. Thus we obtain a contradiction.
EXERCISES

1.1 The following are the state diagrams of two DFAs, $M_1$ and $M_2$. Answer the following questions about each of these machines.

\[ M_1 \quad M_2 \]

\begin{align*}
q_1 & \quad q_1 & \\
\qquad & a & a \\
b & q_2 & b & \\
a, b & q_3 & a, b & \\
\end{align*}

\begin{align*}
q_1 & \quad q_1 & \\
a & q_2 & a & \\
b & q_3 & b & \\
a, b & q_4 & a, b & \\
\end{align*}

a. What is the start state?
b. What is the set of accept states?
c. What sequence of states does the machine go through on input $aabb$?
d. Does the machine accept the string $aabb$?
e. Does the machine accept the string $\epsilon$?

1.2 Give the formal description of the machines $M_1$ and $M_2$ pictured in Exercise 1.1.

1.3 The formal description of a DFA $M$ is $\left( \{q_1, q_2, q_3, q_4, q_5\}, \{u, d\}, \delta, q_3, \{q_3\} \right)$, where $\delta$ is given by the following table. Give the state diagram of this machine.

<table>
<thead>
<tr>
<th>$u$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_5$</td>
<td>$q_4$</td>
</tr>
</tbody>
</table>

1.4 Each of the following languages is the intersection of two simpler languages. In each part, construct DFAs for the simpler languages, then combine them using the construction discussed in footnote 3 (page 46) to give the state diagram of a DFA for the language given. In all parts $\Sigma = \{a, b\}$.

a. $\{w \mid w$ has at least three $a$'s and at least two $b$'s $\}$

b. $\{w \mid w$ has at exactly two $a$'s and at least two $b$'s $\}$

c. $\{w \mid w$ has an even number of $a$'s and one or two $b$'s $\}$

d. $\{w \mid w$ has an even number of $a$'s and each $a$ is followed by at least one $b$ $\}$

e. $\{w \mid w$ starts with an $a$ and has at most one $b$ $\}$

f. $\{w \mid w$ has an odd number of $a$'s and ends with $a$ $b$ $\}$

g. $\{w \mid w$ has even length and an odd number of $a$'s $\}$

1.5 Each of the following languages is the complement of a simpler language. In each part, construct a DFA for the simpler language, then use it to give the state diagram of a DFA for the language given. In all parts \( \Sigma = \{a, b\} \).

\(^\text{Aa.}\) \(\{w \mid w \text{ does not contain the substring } ab\}\)
\(^\text{Ab.}\) \(\{w \mid w \text{ does not contain the substring } baba\}\)
\(^\text{Ac.}\) \(\{w \mid w \text{ contains neither the substrings } ab^* \text{ nor } ba\}\)
\(^\text{Ad.}\) \(\{w \mid w \text{ is any string not in } a^* b^* \}\)
\(^\text{ Ae.}\) \(\{w \mid w \text{ is any string not in } (ab^*)^* \}\)
\(^\text{Af.}\) \(\{w \mid w \text{ is any string not in } a^* \cup b^* \}\)
\(^\text{Ag.}\) \(\{w \mid w \text{ is any string that doesn't contain exactly two } a's\}\)
\(^\text{Ah.}\) \(\{w \mid w \text{ is any string except } a \text{ and } b\}\)

1.6 Give state diagrams of DFAs recognizing the following languages. In all parts the alphabet is \( \{0,1\} \).

\(^\text{a.}\) \(\{w \mid w \text{ begins with a } 1 \text{ and ends with a } 0\}\)
\(^\text{b.}\) \(\{w \mid w \text{ contains at least three } 1's\}\)
\(^\text{c.}\) \(\{w \mid w \text{ contains the substring } 0101, \text{i.e.}, w = x0101y \text{ for some } x \text{ and } y\}\)
\(^\text{d.}\) \(\{w \mid w \text{ has length at least 3 and its third symbol is a } 0\}\)
\(^\text{e.}\) \(\{w \mid w \text{ starts with } 0 \text{ and has odd length, or starts with } 1 \text{ and has even length}\}\)
\(^\text{f.}\) \(\{w \mid w \text{ doesn't contain the substring } 110\}\)
\(^\text{g.}\) \(\{w \mid \text{the length of } w \text{ is at most } 5\}\)
\(^\text{h.}\) \(\{w \mid w \text{ is any string except } 11 \text{ and } 111\}\)
\(^\text{i.}\) \(\{w \mid \text{every odd position of } w \text{ is a } 1\}\)
\(^\text{j.}\) \(\{w \mid w \text{ contains at least two } 0\text{s and at most one } 1\}\)
\(^\text{k.}\) \(\{\varepsilon, 0\}\)
\(^\text{l.}\) \(\{w \mid w \text{ contains an even number of } 0\text{s, or contains exactly two } 1\text{s}\}\)
\(^\text{m.}\) The empty set
\(^\text{n.}\) All strings except the empty string

1.7 Give state diagrams of NFAs with the specified number of states recognizing each of the following languages. In all parts the alphabet is \( \{0,1\} \).

\(^\text{Aa.}\) The language \(\{w \mid w \text{ ends with } 00\}\) with three states
\(^\text{b.}\) The language of Exercise 1.6c with five states
\(^\text{c.}\) The language of Exercise 1.6l with six states
\(^\text{d.}\) The language \(\{0\}\) with two states
\(^\text{e.}\) The language \(0^*1^*0^*\) with three states
\(^\text{Af.}\) The language \(1^*(001^*)^*\) with three states
\(^\text{g.}\) The language \(\{\varepsilon\}\) with one state
\(^\text{h.}\) The language \(0^*\) with one state

1.8 Use the construction given in the proof of Theorem 1.45 to give the state diagrams of NFAs recognizing the union of the languages described in

\(^\text{a.}\) Exercises 1.6a and 1.6b.
\(^\text{b.}\) Exercises 1.6c and 1.6f.
1.9 Use the construction given in the proof of Theorem 1.47 to give the state diagrams of NFAs recognizing the concatenation of the languages described in
   a. Exercises 1.6g and 1.6i.
   b. Exercises 1.6b and 1.6m.

1.10 Use the construction given in the proof of Theorem 1.49 to give the state diagrams of NFAs recognizing the star of the language described in
   a. Exercise 1.6b.
   b. Exercise 1.6i.
   c. Exercise 1.6m.

^1.11 Prove that every NFA can be converted to an equivalent one that has a single accept state.

1.12 Let $D = \{ w \mid w$ contains an even number of a's and an odd number of b's and does not contain the substring ab$\}$. Give a DFA with five states that recognizes $D$ and a regular expression that generates $D$. (Suggestion: Describe $D$ more simply.)

1.13 Let $F$ be the language of all strings over $\{0,1\}$ that do not contain a pair of 1s that are separated by an odd number of symbols. Give the state diagram of a DFA with 5 states that recognizes $F$. (You may find it helpful first to find a 4-state NFA for the complement of $F$.)

1.14 a. Show that, if $M$ is a DFA that recognizes language $B$, swapping the accept and nonaccept states in $M$ yields a new DFA that recognizes the complement of $B$. Conclude that the class of regular languages is closed under complement.
   
   b. Show by giving an example that, if $M$ is an NFA that recognizes language $C$, swapping the accept and nonaccept states in $M$ doesn't necessarily yield a new NFA that recognizes the complement of $C$. Is the class of languages recognized by NFAs closed under complement? Explain your answer.

1.15 Give a counterexample to show that the following construction fails to prove Theorem 1.49, the closure of the class of regular languages under the star operation.\(^8\) Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize $A_1$. Construct $N = (Q_1, \Sigma, \delta, q_1, F)$ as follows. $N$ is supposed to recognize $A_1^*$.  

   a. The states of $N$ are the states of $N_1$.
   b. The start state of $N$ is the same as the start state of $N_1$.
   c. $F = \{ q_1 \} \cup F_1$.
   
   The accept states $F$ are the old accept states plus its start state.
   
   d. Define $\delta$ so that for any $q \in Q$ and any $a \in \Sigma_e$,

   $$\delta(q, a) = \begin{cases} 
   \delta_1(q, a) & q \notin F_1 \text{ or } a \neq \varepsilon \\
   \delta_1(q, a) \cup \{ q_1 \} & q \in F_1 \text{ and } a = \varepsilon.
   \end{cases}$$

   (Suggestion: Show this construction graphically, as in Figure 1.50.)

\(^8\)In other words, you must present a finite automaton, $N_1$, for which the constructed automaton $N$ does not recognize the star of $N_1$'s language.
1.16 Use the construction given in Theorem 1.39 to convert the following two nondeterministic finite automata to equivalent deterministic finite automata.

(a) 

(b) 

1.17  

a. Give an NFA recognizing the language \((01 \cup 001 \cup 010)^*\).

b. Convert this NFA to an equivalent DFA. Give only the portion of the DFA that is reachable from the start state.

1.18 Give regular expressions generating the languages of Exercise 1.6.

1.19 Use the procedure described in Lemma 1.55 to convert the following regular expressions to nondeterministic finite automata.

a. \((0 \cup 1)^*000(0 \cup 1)^*\)

b. \(((00)^*11) \cup 01)^*\)

c. \(\emptyset^*\)

1.20 For each of the following languages, give two strings that are members and two strings that are not members—a total of four strings for each part. Assume the alphabet \(\Sigma = \{a, b\}\) in all parts.

a. \(a^*b^*\)  
e. \(\Sigma^*\Sigma^*b\Sigma^*\Sigma^*\)

b. \(a(ba)^*b\)  
f. \(aba \cup bab\)

c. \(a^* \cup b^*\)  
g. \((\varepsilon \cup a)b\)

d. \((aaa)^*\)  
h. \((a \cup ba \cup bb)^*\)

1.21 Use the procedure described in Lemma 1.60 to convert the following finite automata to regular expressions.
1.22 In certain programming languages, comments appear between delimiters such as /# and #/. Let $C$ be the language of all valid delimited comment strings. A member of $C$ must begin with /# and end with #/ but have no intervening #/. For simplicity, we'll say that the comments themselves are written with only the symbols a and b; hence the alphabet of $C$ is $\Sigma = \{a, b, /, \#\}$.

a. Give a DFA that recognizes $C$.
b. Give a regular expression that generates $C$.

^1.23 Let $B$ be any language over the alphabet $\Sigma$. Prove that $B = B^*$ iff $BB \subseteq B$.

1.24 A finite state transducer (FST) is a type of deterministic finite automaton whose output is a string and not just accept or reject. The following are state diagrams of finite state transducers $T_1$ and $T_2$.

Each transition of an FST is labeled with two symbols, one designating the input symbol for that transition and the other designating the output symbol. The two symbols are written with a slash, /, separating them. In $T_1$, the transition from $q_1$ to $q_2$ has input symbol 2 and output symbol 1. Some transitions may have multiple input–output pairs, such as the transition in $T_1$ from $q_1$ to itself. When an FST computes on an input string $w$, it takes the input symbols $w_1 \cdots w_n$ one by one and, starting at the start state, follows the transitions by matching the input labels with the sequence of symbols $w_1 \cdots w_n = w$. Every time it goes along a transition, it outputs the corresponding output symbol. For example, on input 2212011, machine $T_1$ enters the sequence of states $q_1, q_2, q_2, q_2, q_2, q_1, q_1, q_1$ and produces output 1111000. On input abbb, $T_2$ outputs 1011. Give the sequence of states entered and the output produced in each of the following parts.

a. $T_1$ on input 011
d. $T_1$ on input 0202
e. $T_2$ on input b
f. $T_2$ on input babb
c. $T_1$ on input 121
g. $T_2$ on input bbbbbb
d. $T_1$ on input 0202
h. $T_2$ on input $\varepsilon$

1.25 Read the informal definition of the finite state transducer given in Exercise 1.24. Give a formal definition of this model, following the pattern in Definition 1.5 (page 35). Assume that an FST has an input alphabet $\Sigma$ and an output alphabet $\Gamma$ but not a set of accept states. Include a formal definition of the computation of an FST. (Hint: An FST is a 5-tuple. Its transition function is of the form $\delta: Q \times \Sigma \rightarrow Q \times \Gamma$.)
1.26 Using the solution you gave to Exercise 1.25, give a formal description of the machines $T_1$ and $T_2$ depicted in Exercise 1.24.

1.27 Read the informal definition of the finite state transducer given in Exercise 1.24. Give the state diagram of an FST with the following behavior. Its input and output alphabets are $\{0, 1\}$. Its output string is identical to the input string on the even positions but inverted on the odd positions. For example, on input 0000111 it should output 1010010.

1.28 Convert the following regular expressions to NFAs using the procedure given in Theorem 1.54. In all parts $\Sigma = \{a, b\}$.

a. $a(abb)^* \cup b$

b. $a^* \cup (ab)^*$

c. $(a \cup b^*)a+b^*$

1.29 Use the pumping lemma to show that the following languages are not regular.

A. $A_1 = \{0^n1^n2^n \mid n \geq 0\}$

b. $A_2 = \{www \mid w \in \{a, b\}^*\}$

c. $A_3 = \{a^{2^n} \mid n \geq 0\}$ (Here, $a^{2^n}$ means a string of $2^n$ a's.)

1.30 Describe the error in the following “proof” that $0^*1^*$ is not a regular language. (An error must exist because $0^*1^*$ is regular.) The proof is by contradiction. Assume that $0^*1^*$ is regular. Let $p$ be the pumping length for $0^*1^*$ given by the pumping lemma. Choose $s$ to be the string $0^p1^p$. You know that $s$ is a member of $0^*1^*$, but Example 1.73 shows that $s$ cannot be pumped. Thus you have a contradiction. So $0^*1^*$ is not regular.

---

PROBLEMS

1.31 For any string $w = w_1w_2 \cdots w_n$, the reverse of $w$, written $w^R$, is the string $w$ in reverse order, $w_n \cdots w_2w_1$. For any language $A$, let $A^R = \{w^R \mid w \in A\}$.

Show that if $A$ is regular, so is $A^R$.

1.32 Let

$$\Sigma_3 = \begin{bmatrix} [0] & [0] & [0] \\ [0] & [1] & [0] \\ [1] & [0] & [1] \end{bmatrix}.$$

$\Sigma_3$ contains all size 3 columns of 0s and 1s. A string of symbols in $\Sigma_3$ gives three rows of 0s and 1s. Consider each row to be a binary number and let

$$B = \{w \in \Sigma_3^* \mid \text{the bottom row of } w \text{ is the sum of the top two rows}\}.$$ 

For example,

$$\begin{bmatrix} [0] \\ [0] \\ [1] \end{bmatrix} \in B, \quad \text{but} \quad \begin{bmatrix} [0] \\ [1] \\ [1] \end{bmatrix} \notin B.$$ 

Show that $B$ is regular. (Hint: Working with $B^R$ is easier. You may assume the result claimed in Problem 1.31.)
1.33 Let 
\[ \Sigma_2 = \{ [0], [1], [00], [11] \}. \]
Here, \( \Sigma_2 \) contains all columns of 0s and 1s of height two. A string of symbols in \( \Sigma_2 \) gives two rows of 0s and 1s. Consider each row to be a binary number and let 
\[ C = \{ w \in \Sigma_2 \} \text{ the bottom row of } w \text{ is three times the top row}. \]
For example, \([0][1][1][0] \notin C\), but \([0][0][1][0] \in C\). Show that \( C \) is regular.
(You may assume the result claimed in Problem 1.31.)

1.34 Let \( \Sigma_2 \) be the same as in Problem 1.33. Consider each row to be a binary number and let 
\[ D = \{ w \in \Sigma_2 \text{ the top row of } w \text{ is a larger number than is the bottom row} \}. \]
For example, \([0][1][1][0] \notin D\), but \([0][0][1][0] \notin D\). Show that \( D \) is regular.

1.35 Let \( \Sigma_2 \) be the same as in Problem 1.33. Consider the top and bottom rows to be strings of 0s and 1s and let 
\[ E = \{ w \in \Sigma_2 \text{ the bottom row of } w \text{ is the reverse of the top row of } w \}. \]
Show that \( E \) is not regular.

1.36 Let \( B_n = \{ a^k \} \) where \( k \) is a multiple of \( n \). Show that for each \( n \geq 1 \), the language \( B_n \) is regular.

1.37 Let \( C_n = \{ x \mid x \text{ is a binary number that is a multiple of } n \} \). Show that for each \( n \geq 1 \), the language \( C_n \) is regular.

1.38 An \textbf{all-NFA} \( M \) is a 5-tuple \( (Q, \Sigma, \delta, q_0, F) \) that accepts \( x \in \Sigma^* \) if every possible state that \( M \) could be in after reading input \( x \) is a state from \( F \). Note, in contrast, that an ordinary NFA accepts a string if \textit{some} state among these possible states is an accept state. Prove that all-NFAs recognize the class of regular languages.

1.39 The construction in Theorem 1.54 shows that every GNFA is equivalent to a GNFA with only two states. We can show that an opposite phenomenon occurs for DFAs. Prove that for every \( k > 1 \) a language \( A_k \subseteq \{0,1\}^* \) exists that is recognized by a DFA with \( k \) states but not by one with only \( k - 1 \) states.

1.40 Say that string \( x \) is a \textbf{prefix} of string \( y \) if a string \( z \) exists where \( xz = y \) and that \( x \) is a \textbf{proper prefix} of \( y \) if in addition \( x \neq y \). In each of the following parts we define an operation on a language \( A \). Show that the class of regular languages is closed under that operation.

\( A.\) \( \text{NOPREFIX}(A) = \{ w \in A \mid \text{no proper prefix of } w \text{ is a member of } A \} \).

\( b.\) \( \text{NOEXTEND}(A) = \{ w \in A \mid w \text{ is not the proper prefix of any string in } A \} \).

1.41 For languages \( A \) and \( B \), let the \textbf{perfect shuffle} of \( A \) and \( B \) be the language
\[ \{ w \mid w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma \}. \]
Show that the class of regular languages is closed under perfect shuffle.

1.42 For languages \( A \) and \( B \), let the \textbf{shuffle} of \( A \) and \( B \) be the language
\[ \{ w \mid w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma^* \}. \]
Show that the class of regular languages is closed under shuffle.
1.43 Let \( A \) be any language. Define \( \text{DROP-OUT}(A) \) to be the language containing all strings that can be obtained by removing one symbol from a string in \( A \). Thus, \( \text{DROP-OUT}(A) = \{xz \mid xyz \in A \text{ where } x, z \in \Sigma^*, y \in \Sigma \} \). Show that the class of regular languages is closed under the DROP-OUT operation. Give both a proof by picture and a more formal proof by construction as in Theorem 1.47.

1.44 Let \( B \) and \( C \) be languages over \( \Sigma = \{0, 1\} \). Define
\[
B \upharpoonright C = \{w \in B \mid \text{for some } y \in C, \text{ strings } w \text{ and } y \text{ contain equal numbers of 1s}\}.
\]
Show that the class of regular languages is closed under the \( \upharpoonright \) operation.

1.45 Let \( A/B = \{w \mid wx \in A \text{ for some } x \in B\} \). Show that if \( A \) is regular and \( B \) is any language then \( A/B \) is regular.

1.46 Prove that the following languages are not regular. You may use the pumping lemma and the closure of the class of regular languages under union, intersection, and complement.

a. \( \{0^n1^m0^n \mid m, n \geq 0\} \)

b. \( \{0^m1^n \mid m \neq n\} \)

c. \( \{w \mid w \in \{0,1\}^* \text{ is not a palindrome}\} \)

d. \( \{wtw \mid w, t \in \{0,1\}^*\} \)

1.47 Let \( \Sigma = \{1, \#\} \) and let
\[
Y = \{w \mid w = x_1\#x_2\# \cdots \#x_k \text{ for } k \geq 0, \text{ each } x_i \in 1^*, \text{ and } x_i \neq x_j \text{ for } i \neq j\}.
\]
Prove that \( Y \) is not regular.

1.48 Let \( \Sigma = \{0,1\} \) and let
\[
D = \{w \mid w \text{ contains an equal number of occurrences of the substrings } 01 \text{ and } 10\}.
\]
Thus \( 101 \in D \) because \( 101 \) contains a single \( 01 \) and a single \( 10 \), but \( 1010 \not\in D \) because \( 1010 \) contains two \( 10s \) and one \( 01 \). Show that \( D \) is a regular language.

1.49 a. Let \( B = \{1^k y \mid y \in \{0,1\}^* \text{ and } y \text{ contains at least } k \text{ 1s, for } k \geq 1\} \).
Show that \( B \) is a regular language.

b. Let \( C = \{1^k y \mid y \in \{0,1\}^* \text{ and } y \text{ contains at most } k \text{ 1s, for } k \geq 1\} \).
Show that \( C \) isn’t a regular language.

1.50 Read the informal definition of the finite state transducer given in Exercise 1.24. Prove that no FST can output \( w^R \) for every input \( w \) if the input and output alphabets are \( \{0,1\} \).

1.51 Let \( x \) and \( y \) be strings and let \( L \) be any language. We say that \( x \) and \( y \) are \textit{distinguishable by }\( L \) if some string \( z \) exists whereby exactly one of the strings \( xz \) and \( yz \) is a member of \( L \); otherwise, for every string \( z \), we have \( xz \in L \) whenever \( yz \in L \) and we say that \( x \) and \( y \) are \textit{indistinguishable by }\( L \). If \( x \) and \( y \) are indistinguishable by \( L \) we write \( x \equiv_L y \). Show that \( \equiv_L \) is an equivalence relation.

---

9A \textit{palindrome} is a string that reads the same forward and backward.
A1.52 Myhill–Nerode theorem. Refer to Problem 1.51. Let \( L \) be a language and let \( X \) be a set of strings. Say that \( X \) is pairwise distinguishable by \( L \) if every two distinct strings in \( X \) are distinguishable by \( L \). Define the index of \( L \) to be the maximum number of elements in any set that is pairwise distinguishable by \( L \). The index of \( L \) may be finite or infinite.

a. Show that, if \( L \) is recognized by a DFA with \( k \) states, \( L \) has index at most \( k \).
b. Show that, if the index of \( L \) is a finite number \( k \), it is recognized by a DFA with \( k \) states.
c. Conclude that \( L \) is regular iff it has finite index. Moreover, its index is the size of the smallest DFA recognizing it.

1.53 Let \( \Sigma = \{0, 1, +, =\} \) and

\[
ADD = \{ x = y + z \mid x, y, z \text{ are binary integers, and } x \text{ is the sum of } y \text{ and } z \}.
\]

Show that \( ADD \) is not regular.

1.54 Consider the language \( F = \{ a^i b^j c^k \mid i, j, k \geq 0 \text{ and if } i = 1 \text{ then } j = k \} \).

a. Show that \( F \) is not regular.
b. Show that \( F \) acts like a regular language in the pumping lemma. In other words, give a pumping length \( p \) and demonstrate that \( F \) satisfies the three conditions of the pumping lemma for this value of \( p \).
c. Explain why parts (a) and (b) do not contradict the pumping lemma.

1.55 The pumping lemma says that every regular language has a pumping length \( p \), such that every string in the language can be pumped if it has length \( p \) or more. If \( p \) is a pumping length for language \( A \), so is any length \( p' \geq p \). The minimum pumping length for \( A \) is the smallest \( p \) that is a pumping length for \( A \). For example, if \( A = 01^* \), the minimum pumping length is 2. The reason is that the string \( s = 0 \) is in \( A \) and has length 1 yet \( s \) cannot be pumped, but any string in \( A \) of length 2 or more contains a 1 and hence can be pumped by dividing it so that \( x = 0 \), \( y = 1 \), and \( z \) is the rest. For each of the following languages, give the minimum pumping length and justify your answer.

\[
\begin{align*}
^Aa. \quad & 0001^* \\
^Ab. \quad & 0^*1^* \\
^Ac. \quad & 001 \cup 0^*1^* \\
^Ad. \quad & 0^*1^*0^*1^* \cup 10^*1 \\
^Ae. \quad & (01)^* \\
^Af. \quad & \varepsilon \\
^Ag. \quad & 1^*01^*01^* \\
^Ah. \quad & 10(11^*0)^*0 \\
^Ai. \quad & 1011 \\
^Aj. \quad & \Sigma^*
\end{align*}
\]

1.56 If \( A \) is a set of natural numbers and \( k \) is a natural number greater than 1, let

\[
B_k(A) = \{ w \mid \text{w is the representation in base } k \text{ of some number in } A \}.
\]

Here, we do not allow leading 0s in the representation of a number. For example, \( B_2(\{3, 5\}) = \{11, 101\} \) and \( B_3(\{3, 5\}) = \{10, 12\} \). Give an example of a set \( A \) for which \( B_2(A) \) is regular but \( B_3(A) \) is not regular. Prove that your example works.
1.57 If \( A \) is any language, let \( A\frac{1}{2}_- \) be the set of all first halves of strings in \( A \) so that
\[
A\frac{1}{2}_- = \{ x \mid \text{for some } y, \ |x| = |y| \text{ and } xy \in A \}.
\]
Show that, if \( A \) is regular, then so is \( A\frac{1}{2}_- \).

1.58 If \( A \) is any language, let \( A\frac{1}{3}_- \) be the set of all strings in \( A \) with their middle thirds removed so that
\[
A\frac{1}{3}_- = \{ xz \mid \text{for some } y, \ |x| = |y| = |z| \text{ and } xyz \in A \}.
\]
Show that, if \( A \) is regular, then \( A\frac{1}{3}_- \) is not necessarily regular.

1.59 Let \( M = (Q, \Sigma, \delta, q_0, F) \) be a DFA and let \( h \) be a state of \( M \) called its “home”. A synchronizing sequence for \( M \) and \( h \) is a string \( s \in \Sigma^* \) where \( \delta(q, s) = h \) for every \( q \in Q \). (Here we have extended \( \delta \) to strings, so that \( \delta(q, s) \) equals the state where \( M \) ends up when \( M \) starts at state \( q \) and reads input \( s \).) Say that \( M \) is synchronizable if it has a synchronizing sequence for some state \( h \). Prove that, if \( M \) is a \( k \)-state synchronizable DFA, then it has a synchronizing sequence of length at most \( k^3 \). Can you improve upon this bound?

1.60 Let \( \Sigma = \{ a, b \} \). For each \( k \geq 1 \), let \( C_k \) be the language consisting of all strings that contain an \( a \) exactly \( k \) places from the right-hand end. Thus \( C_k = \Sigma^* a \Sigma^{k-1} \).
Describe an NFA with \( k + 1 \) states that recognizes \( C_k \), both in terms of a state diagram and a formal description.

1.61 Consider the languages \( C_k \) defined in Problem 1.60. Prove that for each \( k \), no DFA can recognize \( C_k \) with fewer than \( 2^k \) states.

1.62 Let \( \Sigma = \{ a, b \} \). For each \( k \geq 1 \), let \( D_k \) be the language consisting of all strings that have at least one \( a \) among the last \( k \) symbols. Thus \( D_k = \Sigma^* a (\Sigma \cup \{ \epsilon \})^{k-1} \).
Describe a DFA with at most \( k + 1 \) states that recognizes \( D_k \), both in terms of a state diagram and a formal description.

1.63 a. Let \( A \) be an infinite regular language. Prove that \( A \) can be split into two infinite disjoint regular subsets.

b. Let \( B \) and \( D \) be two languages. Write \( B \subseteq D \) if \( B \subseteq D \) and \( D \) contains infinitely many strings that are not in \( B \). Show that, if \( B \) and \( D \) are two regular languages where \( B \subseteq D \), then we can find a regular language \( C \) where \( B \subseteq C \subseteq D \).

1.64 Let \( N \) be an NFA with \( k \) states that recognizes some language \( A \).

a. Show that, if \( A \) is nonempty, \( A \) contains some string of length at most \( k \).

b. Show that, by giving an example, that part (a) is not necessarily true if you replace both \( A \)'s by \( A \).

c. Show that, if \( A \) is nonempty, \( A \) contains some string of length at most \( 2^k \).

d. Show that the bound given in part (c) is nearly tight; that is, for each \( k \), demonstrate an NFA recognizing a language \( A_k \) where \( A_k \) is nonempty and where \( A_k \)'s shortest member strings are of length exponential in \( k \). Come as close to the bound in (c) as you can.
1.65 Prove that, for each $n > 0$, a language $B_n$ exists where

a. $B_n$ is recognizable by a NFA that has $n$ states, and
b. if $B_n = A_1 \cup \cdots \cup A_k$, for regular languages $A_i$, then at least one of the $A_i$ requires a DFA with exponentially many states.

SELECTED SOLUTIONS

1.1 For $M_1$: (a) $q_1$; (b) $\{q_2\}$; (c) $q_1, q_2, q_3, q_1, q_1$; (d) No; (e) No
For $M_2$: (a) $q_1$; (b) $\{q_1, q_3\}$; (c) $q_1, q_1, q_2, q_4$; (d) Yes; (e) Yes

1.2 $M_2 = (\{q_1, q_2, q_3\}, \{a, b\}, \delta_1, q_1, \{q_2\})$.

$M_3 = (\{q_1, q_2, q_3, q_4\}, \{a, b\}, \delta_2, q_1, \{q_1, q_4\})$.

The transition functions are

<table>
<thead>
<tr>
<th>$\delta_1$</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$q_2$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_2$</td>
<td>$q_1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\delta_2$</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_4$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_2$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$q_3$</td>
<td>$q_4$</td>
</tr>
</tbody>
</table>

1.4 (b) The following are DFAs for the two languages \{w | w has exactly two a’s\} and \{w | w has at least two b’s\}:

Combining them using the intersection construction gives the DFA:

Though the problem doesn’t request you to simplify the DFA, certain states can be combined to give
(d) These are DFAs for the two languages \( \{ w \mid w \text{ has an even number of a's} \} \) and \( \{ w \mid \text{each a is followed by at least one b} \} \):

Combining them using the intersection construction gives the DFA:

Though the problem doesn't request you to simplify the DFA, certain states can be combined to give
1.5 (a) The left-hand DFA recognizes \( \{ w \mid w \text{ contains } ab \} \). The right-hand DFA recognizes its complement, \( \{ w \mid w \text{ doesn't contain } ab \} \).

(b) This DFA recognizes \( \{ w \mid w \text{ contains baba} \} \).

This DFA recognizes \( \{ w \mid w \text{ does not contain baba} \} \).

1.7 (a) 

1.11 Let \( N = (Q, \Sigma, \delta, q_0, F) \) be any NFA. Construct an NFA \( N' \) with a single accept state that accepts the same language as \( N \). Informally, \( N' \) is exactly like \( N \) except it has \( \varepsilon \)-transitions from the states corresponding to the accept states of \( N \), to a new accept state, \( q_{\text{accept}} \). State \( q_{\text{accept}} \) has no emerging transitions. More formally, \( N' = (Q \cup \{ q_{\text{accept}} \}, \Sigma, \delta', q_0, \{ q_{\text{accept}} \}) \), where for each \( q \in Q \) and \( a \in \Sigma \)

\[
\delta'(q, a) = \begin{cases} 
\delta(q, a) & \text{if } a \neq \varepsilon \text{ or } q \notin F \\
\delta(q, a) \cup \{ q_{\text{accept}} \} & \text{if } a = \varepsilon \text{ and } q \in F
\end{cases}
\]

and \( \delta'(q_{\text{accept}}, a) = \emptyset \) for each \( a \in \Sigma_e \).

1.23 We prove both directions of the "iff."

(\( \rightarrow \)) Assume that \( B = B^* \) and show that \( BB \subseteq B \).
For every language \( BB \subseteq B^* \) holds, so if \( B = B^* \), then \( BB \subseteq B \).

(\( \leftarrow \)) Assume that \( BB \subseteq B \) and show that \( B = B^* \).
For every language \( B \subseteq B^* \), so we need to show only \( B^* \subseteq B \). If \( w \in B^* \), then \( w = x_1 x_2 \cdots x_k \) where each \( x_i \in B \) and \( k \geq 1 \). Because \( x_1, x_2 \in B \) and \( BB \subseteq B \), we have \( x_1 x_2 \in B \). Similarly, because \( x_1 x_2 \) is in \( B \) and \( x_3 \) is in \( B \), we have \( x_1 x_2 x_3 \in B \). Continuing in this way, \( x_1 \cdots x_k \in B \). Hence \( w \in B \), and so we may conclude that \( B^* \subseteq B \).
The latter argument may be written formally as the following proof by induction. Assume that $BB \subseteq B$.

**Claim:** For each $k \geq 1$, if $x_1, \ldots, x_k \in B$, then $x_1 \cdots x_k \in B$.

**Basis:** Prove for $k = 1$. This statement is obviously true.

**Induction step:** For each $k \geq 1$, assume that the claim is true for $k$ and prove it to be true for $k + 1$.

If $x_1, \ldots, x_k, x_{k+1} \in B$, then by the induction assumption, $x_1 \cdots x_k \in B$. Therefore $x_1 \cdots x_k x_{k+1} \in BB$, but $BB \subseteq B$, so $x_1 \cdots x_{k+1} \in B$. That proves the induction step and the claim. The claim implies that, if $BB \subseteq B$, then $B^* \subseteq B$.

1.29 (a) Assume that $A_1 = \{0^n1^n2^n \mid n \geq 0\}$ is regular. Let $p$ be the pumping length given by the pumping lemma. Choose $s$ to be the string $0^p1^p2^p$. Because $s$ is a member of $A_1$ and $s$ is longer than $p$, the pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^iz$ is in $A_1$. Consider two possibilities:

1. The string $y$ consists only of $0$s, only of $1$s, or only of $2$s. In these cases the string $xyyz$ will not have equal numbers of $0$s, $1$s, and $2$s. Hence $xyyz$ is not a member of $A_1$, a contradiction.

2. The string $y$ consists of more than one kind of symbol. In this case $xyyz$ will have the $0$s, $1$s, or $2$s out of order. Hence $xyyz$ is not a member of $A_1$, a contradiction.

Either way we arrive at a contradiction. Therefore, $A_1$ is not regular.

(c) Assume that $A_3 = \{a^{2^n} \mid n \geq 0\}$ is regular. Let $p$ be the pumping length given by the pumping lemma. Choose $s$ to be the string $a^{2^p}$. Because $s$ is a member of $A_1$ and $s$ is longer than $p$, the pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, satisfying the three conditions of the pumping lemma.

The third condition tells us that $|xy| \leq p$. Furthermore, $p < 2^p$ and so $|y| < 2^p$. Therefore $|xyyz| = |xy| + |y| < 2^p + 2^p = 2^{p+1}$. The second condition requires $|y| > 1$ so $2^p < |xyyz| < 2^{p+1}$. The length of $xyyz$ cannot be a power of 2. Hence $xyyz$ is not a member of $A_3$, a contradiction. Therefore, $A_3$ is not regular.

1.40 Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognizing $A$, where $A$ is some regular language. Construct $M' = (Q', \Sigma, \delta', q_0', F')$ recognizing $NOPREFIX(A)$ as follows:

1. $Q' = Q$.
2. For $r \in Q'$ and $a \in \Sigma$ define $\delta'(r, a) = \begin{cases} \delta(r, a) & \text{if } r \notin F \\ \emptyset & \text{if } r \in F. \end{cases}$
3. $q_0' = q_0$.
4. $F' = F$. 
1.44 Let $M_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$ and $M_C = (Q_C, \Sigma, \delta_C, q_C, F_C)$ be DFAs recognizing $B$ and $C$ respectively. Construct NFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes $B \triangleleft C$ as follows. To decide whether its input $w$ is in $B \triangleleft C$, the machine $M$ checks that $w \in B$, and in parallel, nondeterministically guesses a string $y$ that contains the same number of 1s as contained in $w$ and checks that $y \in C$.

1. $Q = Q_B \times Q_C$.
2. For $(q, r) \in Q$ and $a \in \Sigma$ define
   \[
   \delta((q, r), a) = \begin{cases} 
   \{(\delta_B(q, 0), r)\} & \text{if } a = 0 \\
   \{(\delta_B(q, 1), \delta_C(r, 1))\} & \text{if } a = 1 \\
   \{(q, \delta_C(r, 0))\} & \text{if } a = \varepsilon.
   \end{cases}
   \]
3. $q_0 = (q_B, q_C)$.
4. $F = F_B \times F_C$.

1.46 (b) Let $B = \{0^m1^n \mid m \neq n\}$. Observe that $B \cap 0^*1^* = \{0^k1^k \mid k \geq 0\}$. If $B$ were regular, then $B$ would be regular and so would $B \cap 0^*1^*$. But we already know that $\{0^k1^k \mid k \geq 0\}$ isn’t regular, so $B$ cannot be regular.

Alternatively, we can prove $B$ to be nonregular by using the pumping lemma directly, though doing so is trickier. Assume that $B = \{0^m1^n \mid m \neq n\}$ is regular. Let $p$ be the pumping length given by the pumping lemma. Observe that $p!$ is divisible by all integers from 1 to $p$, where $p! = p(p-1)(p-2) \cdots 1$. The string $s = 0^p1^{p+p!} \in B$, and $|s| \geq p$. Thus the pumping lemma implies that $s$ can be divided as $xyz$ with $x = 0^a, y = 0^b$, and $z = 0^{p+p!}$. Let $s’$ be the string $xy^{i+1}z$, where $i = p!/b$. Then $y^i = 0^p$ so $y^{i+1} = 0^{b+p!}$, and so $xyz = 0^{a+b+c+p!} \not\in B$, a contradiction.

1.50 Assume to the contrary that some FST $T$ outputs $w^R$ on input $w$. Consider the input strings 00 and 01. On input 00, $T$ must output 00, and on input 01, $T$ must output 10. In both cases the first input bit is a 0 but the first output bits differ. Operating in this way is impossible for an FST because it produces its first output bit before it reads its second input. Hence no such FST can exist.

1.52 (a) We prove this assertion by contradiction. Let $M$ be a $k$-state DFA that recognizes $L$. Suppose for a contradiction that $L$ has index greater than $k$. That means some set $X$ with more than $k$ elements is pairwise distinguishable by $L$. Because $M$ has $k$ states, the pigeonhole principle implies that $X$ contains two distinct strings $x$ and $y$, where $\delta(q_0, x) = \delta(q_0, y)$. Here $\delta(q_0, x)$ is the state that $M$ is in after starting in the start state $q_0$ and reading input string $x$. Then, for any string $z \in \Sigma^*$, $\delta(q_0, xz) = \delta(q_0, yz)$. Therefore either both $xz$ and $yz$ are in $L$ or neither are in $L$. But then $x$ and $y$ aren’t distinguishable by $L$, contradicting our assumption that $X$ is pairwise distinguishable by $L$.

(b) Let $X = \{s_1, \ldots, s_k\}$ be pairwise distinguishable by $L$. We construct DFA $M = (Q, \Sigma, \delta, q_0, F)$ with $k$ states recognizing $L$. Let $Q = \{q_1, \ldots, q_k\}$, and define $\delta(q_i, a)$ to be $q_j$, where $s_j \equiv_L s_i a$ (the relation $\equiv_L$ is defined in Problem 1.51). Note that $s_j \equiv_L s_i a$ for some $s_j \in X$; otherwise, $X \cup s_i a$ would have $k + 1$ elements and would be pairwise distinguishable by $L$, which would contradict the assumption that $L$ has index $k$. Let $F = \{q_i \mid s_i \in L\}$. Let the start state $q_0$ be the $q_i$ such that $s_i \equiv_L \varepsilon$. $M$ is constructed so that, for any state $q_i$, $\{s \mid \delta(q_0, s) = q_i\} = \{s \mid s \equiv_L s_i\}$. Hence $M$ recognizes $L$. 

SELECTED SOLUTIONS

97
(c) Suppose that \( L \) is regular and let \( k \) be the number of states in a DFA recognizing \( L \). Then from part (a) \( L \) has index at most \( k \). Conversely, if \( L \) has index \( k \), then by part (b) it is recognized by a DFA with \( k \) states and thus is regular. To show that the index of \( L \) is the size of the smallest DFA accepting it, suppose that \( L \)'s index is \textit{exactly} \( k \). Then, by part (b), there is a \( k \)-state DFA accepting \( L \). That is the smallest such DFA because if it were any smaller, then we could show by part (a) that the index of \( L \) is less than \( k \).

1.55  (a) The minimum pumping length is 4. The string 000 is in the language but cannot be pumped, so 3 is not a pumping length for this language. If \( s \) has length 4 or more, it contains 1s. By dividing \( s \) onto \( xyz \), where \( x \) is 000 and \( y \) is the first 1 and \( z \) is everything afterward, we satisfy the pumping lemma's three conditions.

(b) The minimum pumping length is 1. The pumping length cannot be 0 because the string \( \epsilon \) is in the language and it cannot be pumped. Every nonempty string in the language can be divided into \( xyz \), where \( x = \epsilon \) and \( y \) is the first character and \( z \) is the remainder. This division satisfies the three conditions.

(d) The minimum pumping length is 3. The pumping length cannot be 2 because the string 11 is in the language and it cannot be pumped. Let \( s \) be a string in the language of length at least 3. If \( s \) is generated by \( 0^*1^*0^*1^* \), we can write it as \( xyz \), where \( x \) is the empty string, \( y \) is the first symbol of \( s \), and \( z \) is the remainder of \( s \). Breaking \( s \) up in this way shows that it can be pumped. If \( s \) is generated by \( 10^*1 \), we can write it as \( xyz \), where \( x = 1 \) and \( y = 0 \) and \( z \) is the remainder of \( s \). This division gives a way to pump \( s \).