Chapter 3

Regular Expressions and Finite-State Automata

With the set of mathematical tools from the first two chapters, we are now ready to study languages and formal language theory. Our intent is to examine the question of how, and which, languages can be mechanically generated and recognized; and, ultimately, to see what this tells us about what computers can and can’t do.

3.1 Languages

In formal language theory, an alphabet is a finite, non-empty set. The elements of the set are called symbols. A finite sequence of symbols \(a_1a_2\ldots a_n\) from an alphabet is called a string over that alphabet.

Example 3.1. \(\Sigma = \{0, 1\}\) is an alphabet, and 011, 1010, and 1 are all strings over \(\Sigma\).

Note that strings really are sequences of symbols, which implies that order matters. Thus 011, 101, and 110 are all different strings, though they are made up of the same symbols. The strings \(x = a_1a_2\ldots a_n\) and \(y = b_1b_2\ldots b_m\) are equal only if \(m = n\) (i.e. the strings contain the same number of symbols) and \(a_i = b_i\) for all \(1 \leq i \leq n\).

Just as there are operations defined on numbers, truth values, sets, and other mathematical entities, there are operations defined on strings. Some important operations are:
1. **length**: the *length* of a string $x$ is the number of symbols in it. The notation for the length of $x$ is $|x|$. Note that this is consistent with other uses of $| |$, all of which involve some notion of size: $|number|$ measures how big a number is (in terms of its distance from 0); $|set|$ measures the size of a set (in terms of the number of elements).

We will occasionally refer to a *length-n string*. This is a slightly awkward, but concise, shorthand for “a string whose length is n”.

2. **concatenation**: the *concatenation* of two strings $x = a_1a_2\ldots a_m$ and $y = b_1b_2\ldots b_n$ is the sequence of symbols $a_1\ldots a_mb_1\ldots b_n$. Sometimes $\cdot$ is used to denote concatenation, but it is far more usual to see the concatenation of $x$ and $y$ denoted by $xy$ than by $x \cdot y$.

You can easily convince yourself that concatenation is associative (i.e. $(xy)z = x(yz)$ for all strings $x$, $y$, and $z$.) Concatenation is not commutative (i.e. it is not always true that $xy = yx$: for example, if $x = a$ and $y = b$ then $xy = ab$ while $yx = ba$ and, as discussed above, these strings are not equal.)

3. **reversal**: the *reverse* of a string $x = a_1a_2\ldots a_n$ is the string $x^R = a_na_{n-1}\ldots a_2a_1$.

**Example 3.2.** Let $\Sigma = \{a, b\}$, $x = a$, $y = abaa$, and $z = bab$. Then $|x| = 1$, $|y| = 4$, and $|z| = 3$. Also, $xx = aa$, $xy = aabaa$, $xz = abab$, and $zx = bab$. Finally, $x^R = a$, $y^R = aaba$, and $z^R = bab$.

By the way, the previous example illustrates a naming convention standard throughout language theory texts: if a letter is intended to represent a single symbol in an alphabet, the convention is to use a letter from the beginning of the English alphabet ($a$, $b$, $c$, $d$ ); if a letter is intended to represent a string, the convention is to use a letter from the end of the English alphabet ($u$, $v$, etc).

In set theory, we have a special symbol to designate the set that contains no elements. Similarly, language theory has a special symbol $\varepsilon$ which is used to represent the *empty string*, the string with no symbols in it. (Some texts use the symbol $\lambda$ instead.) It is worth noting that $|\varepsilon| = 0$, that $\varepsilon^R = \varepsilon$, and that $\varepsilon \cdot x = x \cdot \varepsilon = x$ for all strings $x$. (This last fact may appear a bit confusing. Remember that $\varepsilon$ is not a symbol in a string with length 1, but rather the name given to the string made up of 0 symbols. Pasting those 0 symbols onto the front or back of a string $x$ still produces $x$.)

The set of all strings over an alphabet $\Sigma$ is denoted $\Sigma^*$. (In language theory, the symbol * is typically used to denote “zero or more”, so $\Sigma^*$ is the
set of strings made up of zero or more symbols from \( \Sigma \).) Note that while an alphabet \( \Sigma \) is by definition a finite set of symbols, and strings are by definition finite sequences of those symbols, the set \( \Sigma^* \) is always infinite. Why is this? Suppose \( \Sigma \) contains \( n \) elements. Then there is one string over \( \Sigma \) with 0 symbols, \( n \) strings with 1 symbol, \( n^2 \) strings with 2 symbols (since there are \( n \) choices for the first symbol and \( n \) choices for the second), \( n^3 \) strings with 3 symbols, etc.

**Example 3.3.** If \( \Sigma = \{1\} \), then \( \Sigma^* = \{\varepsilon, 1, 11, 111, \ldots\} \). If \( \Sigma = \{a, b\} \), then \( \Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aab, aab, \ldots\} \).

Note that \( \Sigma^* \) is countably infinite: if we list the strings as in the preceding example (length-0 strings, length-1 strings in “alphabetical” order, length-2 strings similarly ordered, etc) then any string over \( \Sigma \) will eventually appear. (In fact, if \( |\Sigma| = n \geq 2 \) and \( x \in \Sigma^* \) has length \( k \), then \( x \) will appear on the list within the first \( \frac{2^{k+1}-1}{n-1} \) entries.)

We now come to the definition of a *language* in the formal language theoretical sense.

**Definition 3.1.** A language over an alphabet \( \Sigma \) is a subset of \( \Sigma^* \). Thus, a language over \( \Sigma \) is an element of \( \mathcal{P}(\Sigma) \), the power set of \( \Sigma \).

In other words, any set of strings (over alphabet \( \Sigma \)) constitutes a language (over alphabet \( \Sigma \)).

**Example 3.4.** Let \( \Sigma = \{0, 1\} \). Then the following are all languages over \( \Sigma \):

- \( L_1 = \{011, 1010, 111\} \)
- \( L_2 = \{0, 10, 110, 1110, 11110, \ldots\} \)
- \( L_3 = \{x \in \Sigma^* \mid n_0(x) = n_1(x)\} \), where the notation \( n_0(x) \) stands for the number of 0’s in the string \( x \), and similarly for \( n_1(x) \).
- \( L_4 = \{x \mid x \text{ represents a multiple of 5 in binary}\} \)

Note that languages can be either finite or infinite. Because \( \Sigma^* \) is infinite, it clearly has an infinite number of subsets, and so there are an infinite number of languages over \( \Sigma \). But are there countably or uncountably many such languages?

**Theorem 3.1.** For any alphabet \( \Sigma \), the number of languages over \( \Sigma \) is uncountable.
This fact is an immediate consequence of the result, proved in a previous chapter, that the power set of a countably infinite set is uncountable. Since the elements of $\mathcal{P}(\Sigma)$ are exactly the languages over $\Sigma$, there are uncountably many such languages.

Languages are sets and therefore, as for any sets, it makes sense to talk about the union, intersection, and complement of languages. (When taking the complement of a language over an alphabet $\Sigma$, we always consider the universal set to be $\Sigma^*$, the set of all strings over $\Sigma$.) Because languages are sets of strings, there are additional operations that can be defined on languages, operations that would be meaningless on more general sets. For example, the idea of concatenation can be extended from strings to languages.

For two sets of strings $S$ and $T$, we define the **concatenation** of $S$ and $T$ (denoted $S \cdot T$ or just $ST$) to be the set $ST = \{st \mid s \in S \land t \in T\}$. For example, if $S = \{ab, aab\}$ and $T = \{\varepsilon, 110, 1010\}$, then $ST = \{ab, ab110, ab1010, aab, aab110, aab1010\}$. Note in particular that $ab \in ST$, because $ab \in S$, $\varepsilon \in T$, and $ab \cdot \varepsilon = ab$. Because concatenation of sets is defined in terms of the concatenation of the strings that the sets contain, concatenation of sets is associative and not commutative. (This can easily be verified.)

When a set $S$ is concatenated with itself, the notation $SS$ is usually scrapped in favour of $S^2$; if $S^2$ is concatenated with $S$, we write $S^3$ for the resulting set, etc. So $S^2$ is the set of all strings formed by concatenating two (possibly different, possibly identical) strings from $S$, $S^3$ is the set of strings formed by concatenating three strings from $S$, etc. Extending this notation, we take $S^1$ to be the set of strings formed from one string in $S$ (i.e. $S^1$ is $S$ itself), and $S^0$ to be the set of strings formed from zero strings in $S$ (i.e. $S^0 = \{\varepsilon\}$). If we take the union $S^0 \cup S^1 \cup S^2 \cup \ldots$, then the resulting set is the set of all strings formed by concatenating zero or more strings from $S$, and is denoted $S^*$. The set $S^*$ is called the **Kleene closure** of $S$, and the * operator is called the **Kleene star** operator.

**Example 3.5.** Let $S = \{01, ba\}$. Then

$S^0 = \{\varepsilon\}$

$S^1 = \{01, ba\}$

$S^2 = \{0101, 01ba, ba01, baba\}$

$S^3 = \{010101, 0101ba, 01ba01, 01baba, ba0101, ba01ba, baba01, bababa\}$

etc, so

$S^* = \{\varepsilon, 01, ba, 0101, 01ba, ba01, baba, 010101, 0101ba, \ldots\}$.  

Note that this is the second time we have seen the notation $\text{something}^*$. We have previously seen that for an alphabet $\Sigma$, $\Sigma^*$ is defined to be the set
of all strings over $\Sigma$. If you think of $\Sigma$ as being a set of length-1 strings, and take its Kleene closure, the result is once again the set of all strings over $\Sigma$, and so the two notions of $^*$ coincide.

**Example 3.6.** Let $\Sigma = \{a, b\}$. Then
- $\Sigma^0 = \{\varepsilon\}$
- $\Sigma^1 = \{a, b\}$
- $\Sigma^2 = \{aa, ab, ba, bb\}$
- $\Sigma^3 = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$

etc, so
- $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, \ldots\}$.

**Exercises**

1. Let $S = \{\varepsilon, ab, abab\}$ and $T = \{aa, aba, abba, abbba, \ldots\}$. Find the following:
   a) $S^2$  
   b) $S^3$  
   c) $S^*$  
   d) $ST$  
   e) $TS$

2. The reverse of a language $L$ is defined to be $L^R = \{x^R \mid x \in L\}$. Find $S^R$ and $T^R$ for the $S$ and $T$ in the preceding problem.

3. Give an example of a language $L$ such that $L = L^*$.

### 3.2 Regular Expressions

Though we have used the term *string* throughout to refer to a sequence of symbols from an alphabet, an alternative term that is frequently used is *word*. The analogy seems fairly obvious: strings are made up of “letters” from an alphabet, just as words are in human languages like English. In English, however, there are no particular rules specifying which sequences of letters can be used to form legal English words—even unlikely combinations like *ghth* and *ckstr* have their place. While some formal languages may simply be random collections of arbitrary strings, more interesting languages are those where the strings in the language all share some common structure: $L_1 = \{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$; $L_2 = \{\text{legal Java identifiers}\}$; $L_3 = \{\text{legal C++ programs}\}$. In all of these languages, there are structural rules which determine which sequences of symbols are in the language and which aren’t. So despite the terminology of “alphabet” and “word” in formal language theory, the concepts don’t necessarily match “alphabet” and “word” for human languages. A better parallel is to think of the *alphabet* in a formal language as corresponding to the *words* in a human language; the *words* in a formal language correspond to the *sentences* in a human language, as there are rules (*grammar rules*) which determine how they can legally be constructed.
One way of describing the grammatical structure of the strings in a language is to use a mathematical formalism called a **regular expression**. A regular expression is a pattern that "matches" strings that have a particular form. For example, consider the language (over alphabet $\Sigma = \{a, b\}$) $L = \{x \mid x \text{ starts and ends with } a\}$. What is the symbol-by-symbol structure of strings in this language? Well, they start with an $a$, followed by zero or more $a$’s or $b$’s or both, followed by an $a$. The regular expression $a \cdot (a | b)^* \cdot a$ is a pattern that captures this structure and matches any string in $L$ ($\cdot$ and $^*$ have their usual meanings, and $|$ designates or.$^{1}$) Conversely, consider the regular expression $(a \cdot (a | b)^*) | ((a | b)^* \cdot a)$. This is a pattern that matches any string that either has the form “$a$ followed by zero or more $a$’s or $b$’s or both” (i.e. any string that starts with an $a$) or has the form “zero or more $a$’s or $b$’s or both followed by an $a$” (i.e. any string that ends with an $a$). Thus the regular expression *generates* the language of all strings that start or end (or both) in an $a$: this is the set of strings that match the regular expression.

Here are the formal definitions of a regular expression and the language generated by a regular expression:

**Definition 3.2.** Let $\Sigma$ be an alphabet. Then the following patterns are **regular expressions** over $\Sigma$:

1. $\Phi$ and $\varepsilon$ are regular expressions;
2. $a$ is a regular expression, for each $a \in \Sigma$;
3. if $r_1$ and $r_2$ are regular expressions, then so are $r_1 | r_2$, $r_1 \cdot r_2$, $r_1^*$ and $(r_1)$ (and of course, $r_2^*$ and $(r_2)$). As in concatenation of strings, the $\cdot$ is often left out of the second expression. (Note: the order of precedence of operators, from lowest to highest, is $|$, $\cdot$, $^*$.)

No other patterns are regular expressions.

**Definition 3.3.** The **language generated by a regular expression** $r$, denoted $L(r)$, is defined as follows:

1. $L(\Phi) = \emptyset$, i.e. no strings match $\Phi$;
2. $L(\varepsilon) = \{\varepsilon\}$, i.e. $\varepsilon$ matches only the empty string;
3. $L(a) = \{a\}$, i.e. $a$ matches only the string $a$;

---

$^{1}$Various symbols have been used to represent the “or” operation in regular expressions. Both $+$ and $\cup$ have been used for this purpose. In this book, we use the symbol $|$ because it is commonly used in computer implementations of regular expressions.
4. \( L(r_1 | r_2) = L(r_1) \cup L(r_2) \), i.e. \( r_1 | r_2 \) matches strings that match \( r_1 \) or \( r_2 \) or both;

5. \( L(r_1 r_2) = L(r_1) L(r_2) \), i.e. \( r_1 r_2 \) matches strings of the form “something that matches \( r_1 \) followed by something that matches \( r_2 \)”;

6. \( L(r_1^*) = (L(r_1))^* \), i.e. \( r_1^* \) matches sequences of 0 or more strings each of which matches \( r_1 \).

7. \( L((r_1)) = L(r_1) \), i.e. \( (r_1) \) matches exactly those strings matched by \( r_1 \).

**Example 3.7.** Let \( \Sigma = \{a, b\} \), and consider the regular expression \( r = a^*b^* \). What is \( L(r) \)? Well, \( L(a) = \{a\} \) so \( L(a^*) = (L(a))^* = \{a\}^* \), and \( \{a\}^* \) is the set of all strings of zero or more \( a \)'s, so \( L(a^*) = \{\varepsilon, a, aa, aaa, \ldots\} \). Similarly, \( L(b^*) = \{\varepsilon, b, bb, bbb, \ldots\} \). Since \( L(a^*b^*) = L(a^*) L(b^*) = \{xy \mid x \in L(a^*) \land y \in L(b^*)\} \), we have \( L(a^*b^*) = \{\varepsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb, \ldots\} \), which is the set of all strings of the form “zero or more \( a \)'s followed by zero or more \( b \)'s”.

**Example 3.8.** Let \( \Sigma = \{a, b\} \), and consider the regular expression \( r = (a | aa | aaa)(bb)^* \). Since \( L(a) = \{a\} \), \( L(aa) = L(a) L(a) = \{aa\} \). Similarly, \( L(aaa) = \{aaa\} \) and \( L(bb) = \{bb\} \). Now \( L(a | aa | aaa) = L(a) \cup L(aa) \cup L(aaa) = \{a, aa, aaa\} \), and \( L((bb)^*) = (L((bb))^*) = (L(bb))^* \) (the last equality is from clause 7 of Definition 3.3), and \( L((bb))^* = \{bb\}^* = \{\varepsilon, bb, bbb, \ldots\} \). So \( L(r) \) is the set of strings formed by concatenating \( a \) or \( aa \) or \( aaa \) with zero or more pairs of \( b \)'s.

**Definition 3.4.** A language is regular if it is generated by a regular expression.

Clearly the union of two regular languages is regular; likewise, the concatenation of regular languages is regular; and the Kleene closure of a regular language is regular. It is less clear whether the intersection of regular languages is always regular; nor is it clear whether the complement of a regular language is guaranteed to be regular. These are questions that will be taken up in Section 3.6.

Regular languages, then, are languages whose strings’ structure can be described in a very formal, mathematical way. The fact that a language can be “mechanically” described or generated means that we are likely to be able to get a computer to recognize strings in that language. We will pursue the question of mechanical language recognition in Section 3.4, and subsequently will see that our first attempt to model mechanical language recognition does in fact produce a family of “machines” that recognize
CHAPTER 3. REGULAR EXPRESSIONS AND FSA’S

exactly the regular languages. But first, in the next section, we will look at some practical applications of regular expressions.

Exercises

1. Give English-language descriptions of the languages generated by the following regular expressions.
   a) \((a | b)^*\)   b) \(a^* | b^*\)   c) \(b^* (ab^* ab^*)^*\)   d) \(b^* (abb^*)\)

2. Give regular expressions over \(\Sigma = \{a, b\}\) that generate the following languages.
   a) \(L_1 = \{x \mid x\ \text{contains} \ 3 \ \text{consecutive} \ a's\}\)
   b) \(L_2 = \{x \mid x\ \text{has even length}\}\)
   c) \(L_3 = \{x \mid n_b(x) = 2 \ \text{mod} \ 3\}\)
   d) \(L_4 = \{x \mid x\ \text{contains the substring} \ aaba\}\)
   e) \(L_5 = \{x \mid n_b(x) < 2\}\)
   f) \(L_6 = \{x \mid x\ \text{doesn’t end in} \ aa\}\)

3. Prove that all finite languages are regular.

3.3 Application: Using Regular Expressions

A common operation when editing text is to search for a given string of characters, sometimes with the purpose of replacing it with another string. Many “search and replace” facilities have the option of using regular expressions instead of simple strings of characters. A regular expression describes a language, that is, a set of strings. We can think of a regular expression as a pattern that matches certain strings, namely all the strings in the language described by the regular expression. When a regular expression is used in a search operation, the goal is to find a string that matches the expression. This type of pattern matching is very useful.

The ability to do pattern matching with regular expressions is provided in many text editors, including jedit and kwrite. Programming languages often come with libraries for working with regular expressions. Java (as of version 1.4) provides regular expression handling through a package named java.util.regexp. C++ typically provides a header file named regexp.h for the same purpose. In all these applications, many new notations are added to the syntax to make it more convenient to use. The syntax can vary from one implementation to another, but most implementations include the capabilities discussed in this section.

In applications of regular expressions, the alphabet usually includes all the characters on the keyboard. This leads to a problem, because regular expressions actually use two types of symbols: symbols that are members
of the alphabet and special symbols such as "*" and ")" that are used to construct expressions. These special symbols, which are not part of the language being described but are used in the description, are called **meta-characters**. The problem is, when the alphabet includes all the available characters, what do we do about meta-characters? If the language that we are describing uses the "*" character, for example, how can we represent the Kleene star operation?

The solution is to use a so-called “escape character,” which is usually the backslash, \. We agree, for example, that the notation \* refers to the symbol * that is a member of the alphabet, while * by itself is the meta-character that represents the Kleene star operation. Similarly, ( and ) are the meta-characters that are used for grouping, while the corresponding characters in the language are written as \( and \). For example, a regular expression that matches the string a*b repeated any number of times would be written: (a*\*b)*. The backslash is also used to represent certain non-printing characters. For example, a tab is represented as \t and a new line character is \n.

We introduce two new common operations on regular expressions and two new meta-characters to represent them. The first operation is represented by the meta-character +: If r is a regular expression, then r+ represents the occurrence of r one or more times. The second operation is represented by ?: The notation r? represents an occurrence of r zero or one times. That is to say, r? represents an optional occurrence of r. Note that these operations are introduced for convenience only and do not represent any real increase in the power. In fact, r+ is exactly equivalent to rr*, and r? is equivalent to (r|ε) (except that in applications there is generally no equivalent to ε).

To make it easier to deal with the large number of characters in the alphabet, **character classes** are introduced. A character class consists of a list of characters enclosed between brackets, [ and ]. (The brackets are meta-characters.) A character class matches a single character, which can be any of the characters in the list. For example, [0123456789] matches any one of the digits 0 through 9. The same thing could be expressed as (0123456789), so once again we have added only convenience, not new representational power. For even more convenience, a hyphen can be included in a character class to indicate a range of characters. This means that [0123456789] could also be written as [0-9] and that the regular expression [a-zA-Z] will match any single lowercase letter. A character class can include multiple ranges, so that [a-zA-Z]\* will match any letter, lower- or uppercase. The period (.) is a meta-character that will match any single character, except (in most implementations) for an end-of-line. These
notations can, of course, be used in more complex regular expressions. For example, \([A-Z][a-zA-Z]*\) will match any capitalized word, and \(\(.*\)\) matches any string of characters enclosed in parentheses.

In most implementations, the meta-character ^ can be used in a regular expression to match the beginning of a line of text, so that the expression ^\([a-zA-Z]+\) will only match a word that occurs at the start of a line. Similarly, $ is used as a meta-character to match the end of a line. Some implementations also have a way of matching beginnings and ends of words. Typically, \b will match such “word boundaries.” Using this notation, the pattern \b\textbf{and}\b will match the string “and” when it occurs as a word, but will not match the a-n-d in the word “random.” We are going a bit beyond basic regular expressions here: Previously, we only thought of a regular expression as something that either will match or will not match a given string in its entirety. When we use a regular expression for a search operation, however, we want to find a substring of a given string that matches the expression. The notations ^, $ and \b put a restrictions on where the matching substring can be located in the string.

When regular expressions are used in search-and-replace operations, a regular expression is used for the search pattern. A search is made in a (typically long) string for a substring that matches the pattern, and then the substring is replaced by a specified replacement pattern. The replacement pattern is not used for matching and is not a regular expression. However, it can be more than just a simple string. It’s possible to include parts of the substring that is being replaced in the replacement string. The notations \0, \1, \ldots, \9 are used for this purpose. The first of these, \0, stands for the entire substring that is being replaced. The others are only available when parentheses are used in the search pattern. The notation \1 stands for “the part of the substring that matched the part of the search pattern beginning with the first ( in the pattern and ending with the matching ).” Similarly, \2 represents whatever matched the part of the search pattern between the second pair of parentheses, and so on.

Suppose, for example, that you would like to search for a name in the form \texttt{last-name, first-name} and replace it with the same name in the form \texttt{first-name last-name}. For example, “Reeves, Keanu” should be converted to “Keanu Reeves”. Assuming that names contain only letters, this could be done using the search pattern \(([A-Za-z]+), ([A-Za-z]+)\) and the replacement pattern \2 \1. When the match is made, the first ([A-Za-z]+) will match “Reeves,” so that in the replacement pattern, \1 represents the substring “Reeves”. Similarly, \2 will represent “Keanu”. Note that the parentheses are included in the search pattern only to specify what parts of the string are represented by \1 and \2. In practice, you might use
3.3. APPLICATION: USING REGULAR EXPRESSIONS

^([A-Za-z-]+), ([A-Za-z])$ as the search pattern to constrain it so that it will only match a complete line of text. By using a “global” search-and-replace, you could convert an entire file of names from one format to the other in a single operation.

Regular expressions are a powerful and useful technique that should be part of any computer scientist’s toolbox. This section has given you a taste of what they can do, but you should check out the specific capabilities of the regular expression implementation in the tools and programming languages that you use.

Exercises

1. The backslash is itself a meta-character. Suppose that you want to match a string that contains a backslash character. How do you suppose you would represent the backslash in the regular expression?

2. Using the notation introduced in this section, write a regular expression that could be used to match each of the following:
   a) Any sequence of letters (upper- or lowercase) that includes the letter Z (in uppercase).
   b) Any eleven-digit telephone number written in the form (xxx)xxx-xxxx.
   c) Any eleven-digit telephone number either in the form (xxx)xxx-xxxx or xxx-xxx-xxxx.
   d) A non-negative real number with an optional decimal part. The expression should match numbers such as 17, 183.9999, 182., 0, 0.001, and 21333.2.
   e) A complete line of text that contains only letters.
   f) A C++ style one-line comment consisting of // and all the following characters up to the end-of-line.

3. Give a search pattern and a replace pattern that could be used to perform the following conversions:
   a) Convert a string that is enclosed in a pair of double quotes to the same string with the double quotes replaced by single quotes.
   b) Convert seven-digit telephone numbers in the format xxx-xxx-xxxx to the format (xxx)xxx-xxxx.
   c) Convert C++ one-line comments, consisting of characters between // and end-of-line, to C style comments enclosed between /* and */.
   d) Convert any number of consecutive spaces and tabs to a single space.

4. In some implementations of “regular expressions,” the notations \1, \2, and so on can occur in a search pattern. For example, consider the search pattern ^([a-zA-Z-]+).*$\1$. Here, \1 represents a recurrence of the same substring that matched [a-zA-Z-], the part of the pattern between the first pair of parentheses. The entire pattern, therefore, will match a line of text that
begins and ends with the same letter. Using this notation, write a pattern that matches all strings in the language \( L = \{ a^n b a^n \mid n \geq 0 \} \). (Later in this chapter, we will see that \( L \) is not a regular language, so allowing the use of \( \backslash 1 \) in a “regular expression” means that it’s not really a regular expression at all! This notation can add a real increase in expressive power to the patterns that contain it.)

### 3.4 Finite-State Automata

We have seen how regular expressions can be used to generate languages mechanically. How might languages be recognized mechanically? The question is of interest because if we can mechanically recognize languages like \( L = \{ \text{all legal C++ programs that will not go into infinite loops on any input} \} \), then it would be possible to write über-compilers that can do semantic error-checking like testing for infinite loops, in addition to the syntactic error-checking they currently do.

What formalism might we use to model what it means to recognize a language “mechanically”? We look for inspiration to a language-recognizer with which we are all familiar, and which we’ve already in fact mentioned: a compiler. Consider how a C++ compiler might handle recognizing a legal if statement. Having seen the word if, the compiler will be in a state or phase of its execution where it expects to see a ‘(’; in this state, any other character will put the compiler in a “failure” state. If the compiler does in fact see a ‘(’ next, it will then be in an “expecting a boolean condition” state; if it sees a sequence of symbols that make up a legal boolean condition, it will then be in an “expecting a ‘)’” state; and then “expecting a ‘{’ or a legal statement”; and so on. Thus one can think of the compiler as being in a series of states; on seeing a new input symbol, it moves on to a new state; and this sequence of transitions eventually leads to either a “failure” state (if the if statement is not syntactically correct) or a “success” state (if the if statement is legal). We isolate these three concepts—states, input-inspired transitions from state to state, and “accepting” vs “non-accepting” states—as the key features of a mechanical language-recognizer, and capture them in a model called a finite-state automaton. (Whether this is a successful distillation of the essence of mechanical language recognition remains to be seen; the question will be taken up later in this chapter.)

A finite-state automaton (FSA), then, is a machine which takes, as input, a finite string of symbols from some alphabet \( \Sigma \). There is a finite set of states in which the machine can find itself. The state it is in before consuming any input is called the start state. Some of the states are accepting or final. If the machine ends in such a state after
completely consuming an input string, the string is said to be accepted by the machine. The actual functioning of the machine is described by something called a transition function, which specifies what happens if the machine is in a particular state and looking at a particular input symbol. (“What happens” means “in which state does the machine end up”.)

Example 3.9. Below is a table that describes the transition function of a finite-state automaton with states $p$, $q$, and $r$, on inputs 0 and 1.

<table>
<thead>
<tr>
<th></th>
<th>$p$</th>
<th>$q$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$p$</td>
<td>$q$</td>
<td>$r$</td>
</tr>
<tr>
<td>1</td>
<td>$q$</td>
<td>$r$</td>
<td>$r$</td>
</tr>
</tbody>
</table>

The table indicates, for example, that if the FSA were in state $p$ and consumed a 1, it would move to state $q$.

FSAs actually come in two flavours depending on what properties you require of the transition function. We will look first at a class of FSAs called deterministic finite-state automata (DFAs). In these machines, the current state of the machine and the current input symbol together determine exactly which state the machine ends up in: for every <current state, current input symbol> pair, there is exactly one possible next state for the machine.

Definition 3.5. Formally, a deterministic finite-state automaton $M$ is specified by 5 components: $M = (Q, \Sigma, q_0, \delta, F)$ where

- $Q$ is a finite set of states;
- $\Sigma$ is an alphabet called the input alphabet;
- $q_0 \in Q$ is a state which is designated as the start state;
- $F$ is a subset of $Q$; the states in $F$ are states designated as final or accepting states;
- $\delta$ is a transition function that takes <state, input symbol> pairs and maps each one to a state: $\delta : Q \times \Sigma \rightarrow Q$. To say $\delta(q, a) = q'$ means that if the machine is in state $q$ and the input symbol $a$ is consumed, then the machine will move into state $q'$. The function $\delta$ must be a total function, meaning that $\delta(q, a)$ must be defined for every state $q$ and every input symbol $a$. (Recall also that, according to the definition of a function, there can be only one output for any particular input. This means that for any given $q$ and $a$, $\delta(q, a)$ can
have only one value. This is what makes the finite-state automaton deterministic: given the current state and input symbol, there is only one possible move the machine can make.)

**Example 3.10.** The transition function described by the table in the preceding example is that of a DFA. If we take \( p \) to be the start state and \( r \) to be a final state, then the formal description of the resulting machine is 

\[ M = (\{p, q, r\}, \{0, 1\}, p, \delta, \{r\}) \]

where \( \delta \) is given by

\[
\begin{align*}
\delta(p, 0) &= p & \delta(p, 1) &= q \\
\delta(q, 0) &= q & \delta(q, 1) &= r \\
\delta(r, 0) &= r & \delta(r, 1) &= r
\end{align*}
\]

The transition function \( \delta \) describes only individual steps of the machine as individual input symbols are consumed. However, we will often want to refer to “the state the automaton will be in if it starts in state \( q \) and consumes input string \( w \)”, where \( w \) is a string of input symbols rather than a single symbol. Following the usual practice of using \( * \) to designate “0 or more”, we define \( \delta^*(q, w) \) as a convenient shorthand for “the state that the automaton will be in if it starts in state \( q \) and consumes the input string \( w \)”. For any string, it is easy to see, based on \( \delta \), what steps the machine will make as those symbols are consumed, and what \( \delta^*(q, w) \) will be for any \( q \) and \( w \). Note that if no input is consumed, a DFA makes no move, and so \( \delta^*(q, \epsilon) = q \) for any state \( q \).

**Example 3.11.** Let \( M \) be the automaton in the preceding example. Then, for example:

\[
\begin{align*}
\delta^*(p, 001) &= q, \text{ since } \delta(p, 0) = p, \delta(p, 0) = p, \text{ and } \delta(p, 1) = q; \\
\delta^*(p, 01000) &= q; \\
\delta^*(p, 111) &= r; \\
\delta^*(q, 0010) &= r.
\end{align*}
\]

We have divided the states of a DFA into accepting and non-accepting states, with the idea that some strings will be recognized as “legal” by the automaton, and some not. Formally:

**Definition 3.6.** Let \( M = (Q, \Sigma, q_0, \delta, F) \). A string \( w \in \Sigma^* \) is **accepted** by \( M \) iff \( \delta^*(q_0, w) \in F \). (Don’t get confused by the notation. Remember, it’s just a shorter and neater way of saying “\( w \in \Sigma^* \) is accepted by \( M \))

---

\( \delta^* \) can be defined formally by saying that \( \delta^*(q, \epsilon) = q \) for every state \( q \), and \( \delta^*(q, ax) = \delta^*(\delta(q, a), x) \) for any state \( q, a \in \Sigma \) and \( x \in \Sigma^* \). Note that this is a recursive definition.
and only if the state that $M$ will end up in if it starts in $q_0$ and consumes $w$ is one of the states in $F$."

The language accepted by $M$, denoted $L(M)$, is the set of all strings $w \in \Sigma^*$ that are accepted by $M$: $L(M) = \{ w \in \Sigma^* \mid \delta^*(q_0, w) \in F \}$.

Note that we sometimes use a slightly different phrasing and say that a language $L$ is accepted by some machine $M$. We don’t mean by this that $L$ and maybe some other strings are accepted by $M$; we mean $L = L(M)$, i.e. $L$ is exactly the set of strings accepted by $M$.

It may not be easy, looking at a formal specification of a DFA, to determine what language that automaton accepts. Fortunately, the mathematical description of the automaton $M = (Q, \Sigma, q_0, \delta, F)$ can be neatly and helpfully captured in a picture called a transition diagram. Consider again the DFA of the two preceding examples. It can be represented pictorially as:

The arrow on the left indicates that $p$ is the start state; double circles indicate that a state is accepting. Looking at this picture, it should be fairly easy to see that the language accepted by the DFA $M$ is $L(M) = \{ x \in \{0, 1\}^* \mid n_1(x) \geq 2 \}$.

**Example 3.12.** Find the language accepted by the DFA shown below (and describe it using a regular expression!)

The start state of $M$ is accepting, which means $\varepsilon \in L(M)$. If $M$ is in state $q_0$, a sequence of two $a$’s or three $b$’s will move $M$ back to $q_0$ and hence be accepted. So $L(M) = L((aa \mid bbb)^*)$. 
The state $q_4$ in the preceding example is often called a garbage or trap state: it is a non-accepting state which, once reached by the machine, cannot be escaped. It is fairly common to omit such states from transition diagrams. For example, one is likely to see the diagram:

Note that this cannot be a complete DFA, because a DFA is required to have a transition defined for every state-input pair. The diagram is “short for” the full diagram:

As well as recognizing what language is accepted by a given DFA, we often want to do the reverse and come up with a DFA that accepts a given language. Building DFAs for specified languages is an art, not a science. There is no algorithm that you can apply to produce a DFA from an English-language description of the set of strings the DFA should accept. On the other hand, it is not generally successful, either, to simply write down a half-dozen strings that are in the language and design a DFA to accept those strings—invariably there are strings that are in the language that aren’t accepted, and other strings that aren’t in the language that are accepted. So how do you go about building DFAs that accept all and only the strings they’re supposed to accept? The best advice I can give is to think about relevant characteristics that determine whether a string is in the language or not, and to think about what the possible values or “states” of those characteristics are; then build a machine that has a state corresponding to each possible combination of values of relevant characteristics, and determine how the consumption of inputs affects those values. I’ll illustrate what I mean with a couple of examples.

**Example 3.13.** Find a DFA with input alphabet $\Sigma = \{a, b\}$ that accepts the language $L = \{w \in \Sigma^* \mid n_a(w)$ and $n_b(w)$ are both even $\}$.  

The characteristics that determine whether or not a string $w$ is in $L$ are the parity of $n_a(w)$ and $n_b(w)$. There are four possible combinations of “values” for these characteristics: both numbers could be even, both could be odd, the first could be odd and the second even, or the first could be even
and the second odd. So we build a machine with four states $q_1, q_2, q_3, q_4$ corresponding to the four cases. We want to set up $\delta$ so that the machine will be in state $q_1$ exactly when it has consumed a string with an even number of $a$’s and an even number of $b$’s, in state $q_2$ exactly when it has consumed a string with an odd number of $a$’s and an odd number of $b$’s, and so on.

To do this, we first make the state $q_1$ into our start state, because the DFA will be in the start state after consuming the empty string $\varepsilon$, and $\varepsilon$ has an even number (zero) of both $a$’s and $b$’s. Now we add transitions by reasoning about how the parity of $a$’s and $b$’s is changed by additional input. For instance, if the machine is in $q_1$ (meaning an even number of $a$’s and an even number of $b$’s have been seen) and a further $a$ is consumed, then we want the machine to move to state $q_3$, since the machine has now consumed an odd number of $a$’s and still an even number of $b$’s. So we add the transition $\delta(q_1, a) = q_3$ to the machine. Similarly, if the machine is in $q_2$ (meaning an odd number of $a$’s and an odd number of $b$’s have been seen) and a further $b$ is consumed, then we want the machine to move to state $q_3$ again, since the machine has still consumed an odd number of $a$’s, and now an even number of $b$’s. So we add the transition $\delta(q_2, b) = q_3$ to the machine. Similar reasoning produces a total of eight transitions, one for each state-input pair. Finally, we have to decide which states should be final states. The only state that corresponds to the desired criteria for the language $L$ is $q_1$, so we make $q_1$ a final state. The complete machine is shown below.

![Finite-state automaton diagram]

**Example 3.14.** Find a DFA with input alphabet $\Sigma = \{a, b\}$ that accepts the language $L = \{w \in \Sigma^* \mid n_a(w) \text{ is divisible by 3} \}$.

The relevant characteristic here is of course whether or not the number of $a$’s in a string is divisible by 3, perhaps suggesting a two-state machine. But in fact, there is more than one way for a number to not be divisible by 3: dividing the number by 3 could produce a remainder of either 1 or 2 (a remainder of 0 corresponds to the number in fact being divisible by 3).
So we build a machine with three states $q_0$, $q_1$, $q_2$, and add transitions so that the machine will be in state $q_0$ exactly when the number of $a$’s it has consumed is evenly divisible by 3, in state $q_1$ exactly when the number of $a$’s it has consumed is equivalent to 1 mod 3, and similarly for $q_2$. State $q_0$ will be the start state, as $\varepsilon$ has 0 $a$’s and 0 is divisible by 3. The completed machine is shown below. Notice that because the consumption of $a$ $b$ does not affect the only relevant characteristic, $b$’s do not cause changes of state.

\[ \text{Example 3.15.} \text{ Find a DFA with input alphabet } \Sigma = \{a, b\} \text{ that accepts the language } L = \{w \in \Sigma^* \mid w \text{ contains three consecutive } a\text{'s }\}. \]

Again, it is not quite so simple as making a two-state machine where the states correspond to “have seen $aaa$” and “have not seen $aaa$”. Think dynamically: as you move through the input string, how do you arrive at the goal of having seen three consecutive $a$’s? You might have seen two consecutive $a$’s and still need a third, or you might just have seen one $a$ and be looking for two more to come immediately, or you might just have seen a $b$ and be right back at the beginning as far as seeing 3 consecutive $a$’s goes. So once again there will be three states, with the “last symbol was not an $a$” state being the start state. The complete automaton is shown below.

\[ \text{Exercises} \]

1. Give DFAs that accept the following languages over $\Sigma = \{a, b\}$.
   a) $L_1 = \{x \mid x \text{ contains the substring } aba\}$
   b) $L_2 = L(a^*b^*)$
   c) $L_3 = \{x \mid n_a(x) + n_b(x) \text{ is even} \}$
   d) $L_4 = \{x \mid n_a(x) \text{ is a multiple of } 5 \}$
   e) $L_5 = \{x \mid x \text{ does not contain the substring } abb\}$
3.5. NONDETERMINISTIC FINITE-STATE AUTOMATA

f) \( L_6 = \{ x \mid x \text{ has no } a \text{'s in the even positions} \} \)
g) \( L_7 = L(aa^* | aba* b^*) \)

2. What languages do the following DFAs accept?

a)

b)

3. Let \( \Sigma = \{0, 1\} \). Give a DFA that accepts the language

\[ L = \{ x \in \Sigma^* \mid x \text{ is the binary representation of an integer divisible by 3} \} \].

3.5 Nondeterministic Finite-State Automata

As mentioned briefly above, there is an alternative school of thought as to what properties should be required of a finite-state automaton’s transition function. Recall our motivating example of a C++ compiler and a legal if statement. In our description, we had the compiler in an “expecting a ‘)’” state; on seeing a ‘)’, the compiler moved into an “expecting a ‘{’ or a legal statement” state. An alternative way to view this would be to say that the compiler, on seeing a ‘)’, could move into one of two different states: it could move to an “expecting a ‘{’” state or move to an “expecting a legal statement” state. Thus, from a single state, on input ‘)’, the compiler has multiple moves. This alternative interpretation is not allowed by the DFA model. A second point on which one might question the DFA model is the fact that input must be consumed for the machine to change state. Think of the syntax for C++ function declarations. The return type of a function need not be specified (the default is taken to be \( \text{int} \)). The start state of the compiler when parsing a function declaration might be “expecting a return type”; then with no input consumed, the compiler can move to the state “expecting a legal function name”. To model this, it might seem reasonable to allow transitions that do not require the consumption of input (such
transitions are called \textit{\(\varepsilon\)-transitions}). Again, this is not supported by the DFA abstraction. There is, therefore, a second class of finite-state automata that people study, the class of nondeterministic finite-state automata.

A \textbf{nondeterministic finite-state automaton (NFA)} is the same as a deterministic finite-state automaton except that the transition function is no longer a function that maps a state-input pair to a state; rather, it maps a state-input pair or a state-\(\varepsilon\)-pair to a set of states. No longer do we have \(\delta(q, a) = q'\), meaning that the machine must change to state \(q'\) if it is in state \(q\) and consumes an \(a\). Rather, we have \(\partial(q, a) = \{q_1, q_2, \ldots, q_n\}\), meaning that if the machine is in state \(q\) and consumes an \(a\), it might move directly to any one of the states \(q_1, \ldots, q_n\). Note that the set of next states \(\partial(q, a)\) is defined for every state \(q\) and every input symbol \(a\), but for some \(q\)'s and \(a\)'s it could be empty, or contain just one state (there don't have to be multiple next states). The function \(\partial\) must also specify whether it is possible for the machine to make any moves without input being consumed, i.e. \(\partial(q, \varepsilon)\) must be specified for every state \(q\). Again, it is quite possible that \(\partial(q, \varepsilon)\) may be empty for some states \(q\): there need not be \(\varepsilon\)-transitions out of \(q\).

**Definition 3.7.** Formally, a nondeterministic finite-state automaton \(M\) is specified by 5 components: \(M = (Q, \Sigma, q_0, \partial, F)\) where

- \(Q, \Sigma, q_0\) and \(F\) are as in the definition of DFAs;

- \(\partial\) is a transition function that takes \(<\text{state, input symbol}>\) pairs and maps each one to a set of states. To say \(\partial(q, a) = \{q_1, q_2, \ldots, q_n\}\) means that if the machine is in state \(q\) and the input symbol \(a\) is consumed, then the machine may move directly into any one of states \(q_1, q_2, \ldots, q_n\). The function \(\partial\) must also be defined for every \(<\text{state, } \varepsilon>\) pair. To say \(\partial(q, \varepsilon) = \{q_1, q_2, \ldots, q_n\}\) means that there are direct \(\varepsilon\)-transitions from state \(q\) to each of \(q_1, q_2, \ldots, q_n\).

The formal description of the function \(\partial\) is \(\partial : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)\).

The function \(\partial\) describes how the machine functions on zero or one input symbol. As with DFAs, we will often want to refer to the behavior of the machine on a string of inputs, and so we use the notation \(\partial^*(q, w)\) as shorthand for “the set of states in which the machine might be if it starts in state \(q\) and consumes input string \(w\)”. As with DFAs, \(\partial^*(q, w)\) is determined by the specification of \(\partial\). Note that for every state \(q\), \(\partial^*(q, \varepsilon)\) contains at least \(q\), and may contain additional states if there are (sequences of) \(\varepsilon\)-transitions out of \(q\).
We do have to think a bit carefully about what it means for an NFA to accept a string \( w \). Suppose \( \partial^*(q_0, w) \) contains both accepting and non-accepting states, i.e. the machine could end in an accepting state after consuming \( w \), but it might also end in a non-accepting state. Should we consider the machine to accept \( w \), or should we require every state in \( \partial^*(q_0, w) \) to be accepting before we admit \( w \) to the ranks of the accepted? Think of the C++ compiler again: provided that an if statement fits one of the legal syntax specifications, the compiler will accept it. So we take as the definition of acceptance by an NFA: A string \( w \) is accepted by an NFA provided that at least one of the states in \( \partial^*(q_0, w) \) is an accepting state. That is, if there is some sequence of steps of the machine that consumes \( w \) and leaves the machine in an accepting state, then the machine accepts \( w \).

Formally:

**Definition 3.8.** Let \( M = (Q, \Sigma, q_0, \partial, F) \) be a nondeterministic finite-state automaton. The string \( w \in \Sigma^* \) is **accepted** by \( M \) iff \( \partial^*(q_0, w) \) contains at least one state \( q \in F \).

The **language accepted by** \( M \), denoted \( L(M) \), is the set of all strings \( w \in \Sigma^* \) that are accepted by \( M \): 

\[
L(M) = \{ w \in \Sigma^* | \partial^*(q_0, w) \cap F \neq \emptyset \}.
\]

**Example 3.16.** The NFA shown below accepts all strings of \( a \)'s and \( b \)'s in which the second-to-last symbol is \( a \).

It should be fairly clear that every language that is accepted by a DFA is also accepted by an NFA. Pictorially, a DFA looks exactly like an NFA (an NFA that doesn’t happen to have any \( \varepsilon \)-transitions or multiple same-label transitions from any state), though there is slightly more going on behind the scenes. Formally, given the DFA \( M = (Q, \Sigma, q_0, \delta, F) \), you can build an NFA \( M' = (Q, \Sigma, q_0, \partial, F) \) where 4 of the 5 components are the same and where every transition \( \delta(q, a) = q' \) has been replaced by \( \partial(q, a) = \{q'\} \).

But is the reverse true? Can any NFA-recognized language be recognized by a DFA? Look, for example, at the language in Example 3.16. Can you come up with a DFA that accepts this language? Try it. It’s pretty difficult to do. But does that mean that there really is no DFA that accepts the language, or only that we haven’t been clever enough to find one?

It turns out that the limitation is in fact in our cleverness, and not in the power of DFAs.
Theorem 3.2. Every language that is accepted by an NFA is accepted by a DFA.

Proof. Suppose we are given an NFA \( N = (P, \Sigma, p_0, \delta, F_p) \), and we want to build a DFA \( D = (Q, \Sigma, q_0, \delta, F_q) \) that accepts the same language. The idea is to make the states in \( D \) correspond to subsets of \( N \)'s states, and then to set up \( D \)'s transition function \( \delta \) so that for any string \( w \), \( \delta^*(q_0, w) \) corresponds to \( \partial^*(p_0, w) \); i.e. the single state that \( w \) gets you to in \( D \) corresponds to the set of states that \( w \) could get you to in \( N \). If any of those states is accepting in \( N \), \( w \) would be accepted by \( N \), and so the corresponding state in \( D \) would be made accepting as well.

So how do we make this work? The first thing to do is to deal with a start state \( q_0 \) for \( D \). If we're going to make this state correspond to a subset of \( N \)'s states, what subset should it be? Well, remember (1) that in any DFA, \( \delta^*(q_0, \varepsilon) = q_0 \); and (2) we want to make \( \delta^*(q_0, w) \) correspond to \( \partial^*(p_0, w) \) for every \( w \). Putting these two limitations together tells us that we should make \( q_0 \) correspond to \( \partial^*(p_0, \varepsilon) \). So \( q_0 \) corresponds to the subset of all of \( N \)'s states that can be reached with no input.

Now we progressively set up \( D \)'s transition function \( \delta \) by repeatedly doing the following:

\[-\text{find a state } q \text{ that has been added to } D \text{ but whose out-transitions have not yet been added. (Note that } q_0 \text{ initially fits this description.) Remember that the state } q \text{ corresponds to some subset } \{p_1, \ldots, p_n\} \text{ of } N \text{'s states.}
\]

\[-\text{for each input symbol } a, \text{ look at all } N \text{'s states that can be reached from any one of } p_1, \ldots, p_n \text{ by consuming } a \text{ (perhaps making some } \varepsilon \text{-transitions as well). That is, look at } \partial^*(p_1, a) \cup \ldots \cup \partial^*(p_n, a). \text{ If there is not already a DFA state } q' \text{ that corresponds to this subset of } N \text{'s states, then add one, and add the transition } \delta(q, a) = q' \text{ to } D \text{'s transitions.}
\]

The above process must halt eventually, as there are only a finite number of states \( n \) in the NFA, and therefore there can be at most \( 2^n \) states in the DFA, as that is the number of subsets of the NFA's states. The final states of the new DFA are those where at least one of the associated NFA states is an accepting state of the NFA.

Can we now argue that \( L(D) = L(N) \)? We can, if we can argue that \( \delta^*(q_0, w) \) corresponds to \( \partial^*(p_0, w) \) for all \( w \in \Sigma^* \): if this latter property holds, then \( w \in L(D) \) iff \( \delta^*(q_0, w) \) is accepting, which we make be so iff \( \partial^*(p_0, w) \) contains an accepting state of \( N \), which happens iff \( N \) accepts \( w \) i.e. iff \( w \in L(N) \).

So can we argue that \( \delta^*(q_0, w) \) does in fact correspond to \( \partial^*(p_0, w) \) for all \( w \)? We can, using induction on the length of \( w \).

First, a preliminary observation. Suppose \( w = xa \), i.e. \( w \) is the string \( x \) followed by the single symbol \( a \). How are \( \partial^*(p_0, x) \) and \( \partial^*(p_0, w) \) related?
Well, recall that $\partial^*(p_0, x)$ is the set of all states that $N$ can reach when it starts in $p_0$ and consumes $x$: $\partial^*(p_0, x) = \{p_1, \ldots, p_n\}$ for some states $p_1, \ldots, p_n$. Now, $w$ is just $x$ with an additional $a$, so where might $N$ end up if it starts in $p_0$ and consumes $w$? We know that $x$ gets $N$ to $p_1$ or ... or $p_n$, so $xa$ gets $N$ to any state that can be reached from $p_1$ with an $a$ (and maybe some $\varepsilon$-transitions), and to any state that can be reached from $p_2$ with an $a$ (and maybe some $\varepsilon$-transitions), etc. Thus, our relationship between $\partial^*(p_0, x)$ and $\partial^*(p_0, w)$ is that if $\partial^*(p_0, x) = \{p_1, \ldots, p_n\}$, then $\partial^*(p_0, w) = \partial^*(p_1, a) \cup \ldots \cup \partial^*(p_n, a)$. With this observation in hand, let’s proceed to our proof by induction.

We want to prove that $\delta^*(q_0, w)$ corresponds to $\partial^*(p_0, w)$ for all $w \in \Sigma^*$. We use induction on the length of $w$.

1. Base case: Suppose $w$ has length 0. The only string $w$ with length 0 is $\varepsilon$, so we want to show that $\delta^*(q_0, \varepsilon)$ corresponds to $\partial^*(p_0, \varepsilon)$. Well, $\delta^*(q_0, \varepsilon) = q_0$, since in a DFA, $\delta^*(q, \varepsilon) = q$ for any state $q$. We explicitly made $q_0$ correspond to $\partial^*(p_0, \varepsilon)$, and so the property holds for $w$ with length 0.

2. Inductive case: Assume that the desired property holds for some number $n$, i.e. that $\delta^*(q_0, x)$ corresponds to $\partial^*(p_0, x)$ for all $x$ with length $n$. Look at an arbitrary string $w$ with length $n + 1$. We want to show that $\delta^*(q_0, w)$ corresponds to $\partial^*(p_0, w)$. Well, the string $w$ must look like $xa$ for some string $x$ (whose length is $n$) and some symbol $a$. By our inductive hypothesis, we know $\delta^*(q_0, x)$ corresponds to $\partial^*(p_0, x)$. We know $\partial^*(p_0, x)$ is a set of $N$’s states, say $\partial^*(p_0, x) = \{p_1, \ldots, p_n\}$.

At this point, our subsequent reasoning might be a bit clearer if we give explicit names to $\delta^*(q_0, w)$ (the state $D$ reaches on input $w$) and $\delta^*(q_0, x)$ (the state $D$ reaches on input $x$). Call $\delta^*(q_0, w)$ $q_w$, and call $\delta^*(q_0, x)$ $q_x$. We know, because $w = xa$, there must be an $a$-transition from $q_x$ to $q_w$. Look at how we added transitions to $\delta$: the fact that there is an $a$-transition from $q_x$ to $q_w$ means that $q_w$ corresponds to the set $\partial^*(p_1, a) \cup \ldots \cup \partial^*(p_n, a)$ of $N$’s states. By our preliminary observation, $\partial^*(p_1, a) \cup \ldots \cup \partial^*(p_n, a)$ is just $\partial^*(p_0, w)$. So $q_w$ (or $\delta^*(q_0, w)$) corresponds to $\partial^*(p_0, w)$, which is what we wanted to prove. Since $w$ was an arbitrary string of length $n + 1$, we have shown that the property holds for $n + 1$.

Altogether, we have shown by induction that $\delta^*(q_0, w)$ corresponds to $\partial^*(p_0, w)$ for all $w \in \Sigma^*$. As indicated at the very beginning of this proof, that is enough to prove that $L(D) = L(N)$. So for any NFA $N$, we can find a DFA $D$ that accepts the same language.
Example 3.17. Consider the NFA shown below.

We start by looking at $\partial^*(p_0, \varepsilon)$, and then add transitions and states as described above.

- $\partial^*(p_0, \varepsilon) = \{p_0\}$ so $q_0 = \{p_0\}$.
- $\delta(q_0, a)$ will be $\partial^*(p_0, a)$, which is $\{p_0\}$, so $\delta(q_0, a) = q_0$.
- $\delta(q_0, b)$ will be $\partial^*(p_0, b)$, which is $\{p_0, p_1\}$, so we need to add a new state $q_1 = \{p_0, p_1\}$ to the DFA; and add $\delta(q_0, b) = q_1$ to the DFA’s transition function.

- $\delta(q_1, a)$ will be $\partial^*(p_0, a)$ unioned with $\partial^*(p_1, a)$ since $q_1 = \{p_0, p_1\}$. Since $\partial^*(p_0, a) \cup \partial^*(p_1, a) = \{p_0\} \cup \{p_2\} = \{p_0, p_2\}$, we need to add a new state $q_2 = \{p_0, p_2\}$ to the DFA, and a transition $\delta(q_1, a) = q_2$.
- $\delta(q_1, b)$ will be $\partial^*(p_0, b)$ unioned with $\partial^*(p_1, b)$, which gives $\{p_0, p_1\} \cup \{p_2\}$, which again gives us a new state $q_3$ to add to the DFA, together with the transition $\delta(q_1, b) = q_3$.

At this point, our partially-constructed DFA looks as shown below:

The construction continues as long as there are new states being added, and new transitions from those states that have to be computed. The final DFA is shown below.
3.6. FINITE-STATE AUTOMATA AND REGULAR LANGUAGES

Exercises

1. What language does the NFA in Example 3.17 accept?
2. Give a DFA that accepts the language accepted by the following NFA.

3. Give a DFA that accepts the language accepted by the following NFA. (Be sure to note that, for example, it is possible to reach both $q_1$ and $q_3$ from $q_0$ on consumption of an $a$, because of the $\varepsilon$-transition.)

3.6 Finite-State Automata and Regular Languages

We know now that our two models for mechanical language recognition actually recognize the same class of languages. The question still remains: do they recognize the same class of languages as the class generated mechanically by regular expressions? The answer turns out to be “yes”. There are two parts to proving this: first that every language generated can be recognized, and second that every language recognized can be generated.

Theorem 3.3. Every language generated by a regular expression can be recognized by an NFA.

Proof. The proof of this theorem is a nice example of a proof by induction on the structure of regular expressions. The definition of regular expression is inductive: $\Phi$, $\varepsilon$, and $a$ are the simplest regular expressions, and then more
complicated regular expressions can be built from these. We will show that there are NFAs that accept the languages generated by the simplest regular expressions, and then show how those machines can be put together to form machines that accept languages generated by more complicated regular expressions.

Consider the regular expression $\Phi$. $L(\Phi) = \{\}$. Here is a machine that accepts $\{\}$:

```
  \[
  \begin{array}{ccc}
  & \longrightarrow & \\
  \end{array}
  \]
```

Consider the regular expression $\varepsilon$. $L(\varepsilon) = \{\varepsilon\}$. Here is a machine that accepts $\{\varepsilon\}$:

```
  \[
  \begin{array}{ccc}
  & \longrightarrow & \\
  \end{array}
  \]
```

Consider the regular expression $a$. $L(a) = \{a\}$. Here is a machine that accepts $\{a\}$:

```
  \[
  \begin{array}{ccc}
  & \longrightarrow & a \\
  \end{array}
  \]
```

Now suppose that you have NFAs that accept the languages generated by the regular expressions $r_1$ and $r_2$. Building a machine that accepts $L(r_1 \mid r_2)$ is fairly straightforward: take an NFA $M_1$ that accepts $L(r_1)$ and an NFA $M_2$ that accepts $L(r_2)$. Introduce a new state $q_{\text{new}}$, connect it to the start states of $M_1$ and $M_2$ via $\varepsilon$-transitions, and designate it as the start state of the new machine. No other transitions are added. The final states of $M_1$ together with the final states of $M_2$ are designated as the final states of the new machine. It should be fairly clear that this new machine accepts exactly those strings accepted by $M_1$ together with those strings accepted by $M_2$: any string $w$ that was accepted by $M_1$ will be accepted by the new NFA by starting with an $\varepsilon$-transition to the old start state of $M_1$ and then following the accepting path through $M_1$; similarly, any string accepted by $M_2$ will be accepted by the new machine; these are the only strings that will be accepted by the new machine, as on any input $w$ all the new machine can do is make an $\varepsilon$-move to $M_1$'s (or $M_2$'s) start state, and from there $w$ will only be accepted by the new machine if it is accepted by $M_1$ (or $M_2$). Thus, the new machine accepts $L(M_1) \cup L(M_2)$, which is $L(r_1) \cup L(r_2)$, which is exactly the definition of $L(r_1 \mid r_2)$. 
3.6. FINITE-STATE AUTOMATA AND REGULAR LANGUAGES

(A pause before we continue: note that for the simplest regular expressions, the machines that we created to accept the languages generated by the regular expressions were in fact DFAs. In our last case above, however, we needed \( \varepsilon \)-transitions to build the new machine, and so if we were trying to prove that every regular language could be accepted by a DFA, our proof would be in trouble. THIS DOES NOT MEAN that the statement “every regular language can be accepted by a DFA” is false, just that we can’t prove it using this kind of argument, and would have to find an alternative proof.)

Suppose you have machines \( M_1 \) and \( M_2 \) that accept \( L(r_1) \) and \( L(r_2) \) respectively. To build a machine that accepts \( L(r_1) \cup L(r_2) \) proceed as follows. Make the start state \( q_{01} \) of \( M_1 \) be the start state of the new machine. Make the final states of \( M_2 \) be the final states of the new machine. Add \( \varepsilon \)-transitions from the final states of \( M_1 \) to the start state \( q_{02} \) of \( M_2 \).

It should be fairly clear that this new machine accepts exactly those strings of the form \( xy \) where \( x \in L(r_1) \) and \( y \in L(r_2) \): first of all, any string of this form will be accepted because \( x \in L(r_1) \) implies there is a path that consumes \( x \) from \( q_{01} \) to a final state of \( M_1 \); a \( \varepsilon \)-transition moves to \( q_{02} \); then \( y \in L(r_2) \) implies there is a path that consumes \( y \) from \( q_{02} \) to a final state of \( M_2 \); and the final states of \( M_2 \) are the final states of the new machine, so \( xy \) will be accepted. Conversely, suppose \( z \) is accepted by the new machine. Since the only final states of the new machine are in the old \( M_2 \), and the only way to get into \( M_2 \) is to take a \( \varepsilon \)-transition from a final state of \( M_1 \), this means that \( z = xy \) where \( x \) takes the machine from its...
start state to a final state of \( M_1 \), a \( \varepsilon \)-transition occurs, and then \( y \) takes the machine from \( q_{02} \) to a final state of \( M_2 \). Clearly, \( x \in L(r_1) \) and \( y \in L(r_2) \).

We leave the construction of an NFA that accepts \( L(r^*) \) from an NFA that accepts \( L(r) \) as an exercise.

\[ \square \]

**Theorem 3.4.** Every language that is accepted by a DFA or an NFA is generated by a regular expression.

Proving this result is actually fairly involved and not very illuminating. Before presenting a proof, we will give an illustrative example of how one might actually go about extracting a regular expression from an NFA or a DFA. You can go on to read the proof if you are interested.

**Example 3.18.** Consider the DFA shown below:

![DFA Diagram]

Note that there is a loop from state \( q_2 \) back to state \( q_2 \): any number of \( a \)'s will keep the machine in state \( q_2 \), and so we label the transition with the regular expression \( a^* \). We do the same thing to the transition labeled \( b \) from \( q_0 \). (Note that the result is no longer a DFA, but that doesn’t concern us, we’re just interested in developing a regular expression.)

![DFA Diagram with Regular Expressions]

Next we note that there is in fact a loop from \( q_1 \) to \( q_1 \) via \( q_0 \). A regular expression that matches the strings that would move around the loop is \( ab^*a \). So we add a transition labeled \( ab^*a \) from \( q_1 \) to \( q_1 \), and remove the now-irrelevant \( a \)-transition from \( q_1 \) to \( q_0 \). (It is irrelevant because it is not part of any other loop from \( q_1 \) to \( q_1 \).)
Next we note that there is also a loop from $q_1$ to $q_1$ via $q_2$. A regular expression that matches the strings that would move around the loop is $ba^*b$. Since the transitions in the loop are the only transitions to or from $q_2$, we simply remove $q_2$ and replace it with a transition from $q_1$ to $q_1$.

It is now clear from the diagram that strings of the form $b^*a$ get you to state $q_1$, and any number of repetitions of strings that match $ab^*a$ or $ba^*b$ will keep you there. So the machine accepts $L(b^*a(ab^*a|ba^*b)^*)$.

**Proof of Theorem 3.4.** We prove that the language accepted by a DFA is regular. The proof for NFAs follows from the equivalence between DFAs and NFAs.

Suppose that $M$ is a DFA, where $M = (Q, \Sigma, q_0, \delta, F)$. Let $n$ be the number of states in $M$, and write $Q = \{q_0, q_1, \ldots, q_{n-1}\}$. We want to consider computations in which $M$ starts in some state $q_i$ and reads a string $w$, and ends in state $q_k$. In such a computation, $M$ might go through a series of intermediates states between $q_i$ and $q_k$:

$$q_i \rightarrow p_1 \rightarrow p_2 \cdots \rightarrow p_r \rightarrow q_k$$

We are interested in computations in which all of the intermediate states—$p_1, p_2, \ldots, p_r$—are in the set $\{q_0, q_1, \ldots, q_{j-1}\}$, for some number $j$. We define $R_{i,j,k}$ to be the set of all strings $w$ in $\Sigma^*$ that are consumed by such a computation. That is, $w \in R_{i,j,k}$ if and only if when $M$ starts in state $q_i$ and reads $w$, it ends in state $q_k$, and all the intermediate states between $q_i$ and $q_k$ are in the set $\{q_0, q_1, \ldots, q_{j-1}\}$. $R_{i,j,k}$ is a language over $\Sigma$. We show that $R_{i,j,k}$ is regular for $0 \leq i < n$, $0 \leq j \leq n$, $0 \leq k < n$.

Consider the language $R_{i,0,k}$. For $w \in R_{i,0,k}$, the set of allowable intermediate states is empty. Since there can be no intermediate states, it follows that there can be at most one step in the computation that starts in state $q_i$, reads $w$, and ends in state $q_k$. So, $|w|$ can be at most one. This means that $R_{i,0,k}$ is finite, and hence is regular. (In fact, $R_{i,0,k} = \{a \in \Sigma \mid \delta(q_i, a) = q_k\}$, for $i \neq k$, and $R_{i,0,i} = \{\varepsilon\} \cup \{a \in \Sigma \mid \delta(q_i, a) = q_i\}$. Note that in many cases, $R_{i,0,k}$ will be the empty set.)

We now proceed by induction on $j$ to show that $R_{i,j,k}$ is regular for all $i$ and $k$. We have proved the base case, $j = 0$. Suppose that $0 \leq j < n$. We have now proved that $R_{i,j,k}$ is regular for all $i$, $k$, and $j$. Therefore, $L(b^*a(ab^*a|ba^*b)^*)$ is regular.
we already know that $R_{i,j,k}$ is regular for all $i$ and all $k$. We need to show that $R_{i,j+1,k}$ is regular for all $i$ and $k$. In fact,

$$R_{i,j+1,k} = R_{i,j,k} \cup (R_{i,j,j} R_{j,j,j}^* R_{j,j,k})$$

which is regular because $R_{i,j,k}$ is regular for all $i$ and $k$, and because the union, concatenation, and Kleene star of regular languages are regular.

To see that the above equation holds, consider a string $w \in \Sigma^*$. Now, $w \in R_{i,j+1,k}$ if and only if when $M$ starts in state $q_i$ and reads $w$, it ends in state $q_k$, with all intermediate states in the computation in the set $\{q_0, q_1, \ldots, q_j\}$. Consider such a computation. There are two cases: Either $q_j$ occurs as an intermediate state in the computation, or it does not. If it does not occur, then all the intermediate states are in the set $\{q_0, q_1, \ldots, q_{j-1}\}$, which means that in fact $w \in R_{i,j,k}$. If $q_j$ does occur as an intermediate state in the computation, then we can break the computation into phases, by dividing it at each point where $q_j$ occurs as an intermediate state. This breaks $w$ into a concatenation $w = x y_1 y_2 \cdots y_r z$. The string $x$ is consumed in the first phase of the computation, during which $M$ goes from state $q_i$ to the first occurrence of $q_j$; since the intermediate states in this computation are in the set $\{q_0, q_1, \ldots, q_{j-1}\}$, $x \in R_{i,j,j}$. The string $z$ is consumed by the last phase of the computation, in which $M$ goes from the final occurrence of $q_j$ to $q_k$, so that $z \in R_{j,j,k}$. And each string $y_t$ is consumed in a phase of the computation in which $M$ goes from one occurrence of $q_j$ to the next occurrence of $q_j$, so that $y_r \in R_{j,j,j}$. This means that $w = x y_1 y_2 \cdots y_r z \in R_{i,j,j} R_{j,j,j}^* R_{j,j,k}$.

We now know, in particular, that $R_{0,n,k}$ is a regular language for all $k$. But $R_{0,n,k}$ consists of all strings $w \in \Sigma^*$ such that when $M$ starts in state $q_0$ and reads $w$, it ends in state $q_k$ (with no restriction on the intermediate states in the computation, since every state of $M$ is in the set $\{q_0, q_1, \ldots, q_{n-1}\}$). To finish the proof that $L(M)$ is regular, it is only necessary to note that

$$L(M) = \bigcup_{q_k \in F} R_{0,n,k}$$

which is regular since it is a union of regular languages. This equation is true since a string $w$ is in $L(M)$ if and only if when $M$ starts in state $q_0$ and reads $w$, it ends in some accepting state $q_k \in F$. This is the same as saying $w \in R_{0,n,k}$ for some $k$ with $q_k \in F$.

We have already seen that if two languages $L_1$ and $L_2$ are regular, then so are $L_1 \cup L_2$, $L_1 L_2$, and $L_1^*$ (and of course $L_2^*$). We have not yet seen, however, how the common set operations intersection and complementation
3.6. FINITE-STATE AUTOMATA AND REGULAR LANGUAGES

affect regularity. Is the complement of a regular language regular? How about the intersection of two regular languages?

Both of these questions can be answered by thinking of regular languages in terms of their acceptance by DFAs. Let’s consider first the question of complementation. Suppose we have an arbitrary regular language \( L \). We know there is a DFA \( M \) that accepts \( L \). Pause a moment and try to think of a modification that you could make to \( M \) that would produce a new machine \( M' \) that accepts \( \overline{L} \). Okay, the obvious thing to try is to make \( M' \) be a copy of \( M \) with all final states of \( M \) becoming non-final states of \( M' \) and vice versa. This is in fact exactly right: \( M' \) does in fact accept \( \overline{L} \).

To verify this, consider an arbitrary string \( w \). The transition functions for the two machines \( M \) and \( M' \) are identical, so \( \delta^*(q_0, w) \) is the same state in both \( M \) and \( M' \); if that state is accepting in \( M \) then it is non-accepting in \( M' \), so if \( w \) is accepted by \( M \) it is not accepted by \( M' \); if the state is non-accepting in \( M \) then it is accepting in \( M' \), so if \( w \) is not accepted by \( M \) then it is accepted by \( M' \). Thus \( M' \) accepts exactly those strings that \( M \) does not, and hence accepts \( \overline{L} \).

It is worth pausing for a moment and looking at the above argument a bit longer. Would the argument have worked if we had looked at an arbitrary language \( L \) and an arbitrary NFA \( M \) that accepted \( L \)? That is, if we had built a new machine \( M' \) in which the final and non-final states had been switched, would the new NFA \( M' \) accept the complement of the language accepted by \( M \)? The answer is “not necessarily”. Remember that acceptance in an NFA is determined based on whether or not at least one of the states reached by a string is accepting. So any string \( w \) with the property that \( \partial^*(q_0, w) \) contains both accepting and non-accepting states of \( M \) would be accepted both by \( M \) and by \( M' \).

Now let’s turn to the question of intersection. Given two regular languages \( L_1 \) and \( L_2 \), is \( L_1 \cap L_2 \) regular? Again, it is useful to think in terms of DFAs: given machines \( M_1 \) and \( M_2 \) that accept \( L_1 \) and \( L_2 \), can you use them to build a new machine that accepts \( L_1 \cap L_2 \)? The answer is yes, and the idea behind the construction bears some resemblance to that behind the NFA-to-DFA construction. We want a new machine where transitions reflect the transitions of both \( M_1 \) and \( M_2 \) simultaneously, and we want to accept a string \( w \) only if that those sequences of transitions lead to final states in both \( M_1 \) and \( M_2 \). So we associate the states of our new machine \( M \) with pairs of states from \( M_1 \) and \( M_2 \). For each state \( (q_1, q_2) \) in the new machine and input symbol \( a \), define \( \delta((q_1, q_2), a) \) to be the state \( (\delta_1(q_1, a), \delta_2(q_2, a)) \). The start state \( q_0 \) of \( M \) is \((q_{01}, q_{02})\), where \( q_{0i} \) is the start state of \( M_i \). The final states of \( M \) are the the states of the form \((q_{f1}, q_{f2})\) where \( q_{f1} \) is an accepting state of \( M_1 \) and \( q_{f2} \) is an accepting
state of $M_2$. You should convince yourself that $M$ accepts a string $x$ iff $x$ is accepted by both $M_1$ and $M_2$.

The results of the previous section and the preceding discussion are summarized by the following theorem:

**Theorem 3.5.** The intersection of two regular languages is a regular language.

The union of two regular languages is a regular language.

The concatenation of two regular languages is a regular language.

The complement of a regular language is a regular language.

The Kleene closure of a regular language is a regular language.

**Exercises**

1. Give a DFA that accepts the intersection of the languages accepted by the machines shown below. (Suggestion: use the construction discussed in the chapter just before Theorem 3.5.)

2. Complete the proof of Theorem 3.3 by showing how to modify a machine that accepts $L(r)$ into a machine that accepts $L(r^*)$.

3. Using the construction described in Theorem 3.3, build an NFA that accepts $L((ab|a)^*(bb))$.

4. Prove that the reverse of a regular language is regular.

5. Show that for any DFA or NFA, there is an NFA with exactly one final state that accepts the same language.

6. Suppose we change the model of NFAs to allow NFAs to have multiple start states. Show that for any “NFA” with multiple start states, there is an NFA with exactly one start state that accepts the same language.

7. Suppose that $M_1 = (Q_1, \Sigma, q_1, \delta_1, F_1)$ and $M_2 = (Q_2, \Sigma, q_2, \delta_2, F_2)$ are DFAs over the alphabet $\Sigma$. It is possible to construct a DFA that accepts the language $L(M_1) \cap L(M_2)$ in a single step. Define the DFA

\[ M = (Q_1 \times Q_2, \Sigma, (q_1, q_2), \delta, F_1 \times F_2) \]

where $\delta$ is the function from $(Q_1 \times Q_2) \times \Sigma$ to $Q_1 \times Q_2$ that is defined by:

\[ \delta((p_1, p_2), \sigma) = (\delta_1(p_1, \sigma), \delta_2(p_2, \sigma)) \]

Convince yourself that this definition makes sense. (For example, note that states in $M$ are pairs $(p_1, p_2)$ of states, where $p_1 \in Q_1$ and $p_2 \in Q_2$, and note that the start state $(q_1, q_2)$ in $M$ is
in fact a state in $M$.) Prove that $L(M) = L(M_1) \cap L(M_2)$, and explain why this shows that the intersection of any two regular languages is regular. This proof—if you can get past the notation—is more direct than the one outlined above.

3.7 Non-regular Languages

The fact that our models for mechanical language-recognition accept exactly the same languages as those generated by our mechanical language-generation system would seem to be a very positive indication that in “regular” we have in fact managed to isolate whatever characteristic it is that makes a language “mechanical”. Unfortunately, there are languages that we intuitively think of as being mechanically-recognizable (and which we could write C++ programs to recognize) that are not in fact regular.

How does one prove that a language is not regular? We could try proving that there is no DFA or NFA that accepts it, or no regular expression that generates it, but this kind of argument is generally rather difficult to make. It is hard to rule out all possible automata and all possible regular expressions. Instead, we will look at a property that all regular languages have; proving that a given language does not have this property then becomes a way of proving that that language is not regular.

Consider the language $L = \{ w \in \{a,b\}^* \mid n_a(w) = 2 \mod 3, n_b(w) = 2 \mod 3 \}$. Below is a DFA that accepts this language, with states numbered 1 through 9.

Consider the sequence of states that the machine passes through while processing the string $abbbabb$. Note that there is a repeated state (state 2). We say that $abbbabb$ “goes through the state 2 twice”, meaning that in the course of the string being processed, the machine is in state 2 twice (at
least). Call the section of the string that takes you around the loop \( y \), the preceding section \( x \), and the rest \( z \). Then \( xz \) is accepted, \( xyyz \) is accepted, \( xyyyz \) is accepted, etc. Note that the string \( aabb \) cannot be divided this way, because it does not go through the same state twice. Which strings can be divided this way? Any string that goes through the same state twice. This may include some relatively short strings and must include any string with length greater than or equal to 9, because there are only 9 states in the machine, and so repetition must occur after 9 input symbols at the latest.

More generally, consider an arbitrary DFA \( M \), and let the number of states in \( M \) be \( n \). Then any string \( w \) that is accepted by \( M \) and has \( n \) or more symbols must go through the same state twice, and can therefore be broken up into three pieces \( x, y, z \) (where \( y \) contains at least one symbol) so that \( w = xyz \) and

- \( xz \) is accepted by \( M \)
- \( xyyz \) is accepted by \( M \) (after all, we started with \( w \) in \( L(M) \))
- \( xyyyz \) is accepted by \( M \)

etc.

Note that you can actually say even more: within the first \( n \) characters of \( w \) you must already get a repeated state, so you can always find an \( x, y, z \) as described above where, in addition, the \( xy \) portion of \( w \) (the portion of \( w \) that takes you to and back to a repeated state) contains at most \( n \) symbols.

So altogether, if \( M \) is an \( n \)-state DFA that accepts \( L \), and \( w \) is a string in \( L \) whose length is at least \( n \), then \( w \) can be broken down into three pieces \( x, y, z \), where \( w = xyz \), such that

(i) \( x \) and \( y \) together contain no more than \( n \) symbols;
(ii) \( y \) contains at least one symbol;
(iii) \( xz \) is accepted by \( M \)

\( (xyyz \) is accepted by \( M \))

etc.

The usually-stated form of this result is the Pumping Lemma:

**Theorem 3.6.** If \( L \) is a regular language, then there is some number \( n > 0 \) such that any string \( w \) in \( L \) whose length is greater than or equal to \( n \) can be broken down into three pieces \( x, y, \) and \( z \), \( w = xyz \), such that

(i) \( x \) and \( y \) together contain no more than \( n \) symbols;

(ii) \( y \) contains at least one symbol;

(iii) \( xz \) is accepted by \( M \)
3.7. NON-REGULAR LANGUAGES

\[(xyz \text{ is accepted by } M)\]
\[xyyz \text{ is accepted by } M\]
\[\text{etc.}\]

Though the Pumping Lemma says something about regular languages, it is not used to prove that languages are regular. It says “if a language is regular, then certain things happen”, not “if certain things happen, then you can conclude that the language is regular.” However, the Pumping Lemma is useful for proving that languages are not regular, since the contrapositive of “if a language is regular then certain things happen” is “if certain things don’t happen then you can conclude that the language is not regular.” So what are the “certain things”? Basically, the P.L. says that if a language is regular, there is some “threshold” length for strings, and every string that goes over that threshold can be broken down in a certain way. Therefore, if we can show that “there is some threshold length for strings such that every string that goes over that threshold can be broken down in a certain way” is a false assertion about a language, we can conclude that the language is not regular. How do you show that there is no threshold length? Saying a number is a threshold length for a language means that every string in the language that is at least that long can be broken down in the ways described. So to show that a number is not a threshold value, we have to show that there is some string in the language that is at least that long that cannot be broken down in the appropriate way.

**Theorem 3.7.** \[\{a^n b^n \mid n \geq 0\}\] is not regular.

**Proof.** We do this by showing that there is no threshold value for the language. Let \(N\) be an arbitrary candidate for threshold value. We want to show that it is not in fact a threshold value, so we want to find a string in the language whose length is at least \(N\) and which can’t be broken down in the way described by the Pumping Lemma. What string should we try to prove unbreakable? We can’t pick strings like \(a^{100} b^{100}\) because we’re working with an arbitrary \(N\) i.e. making no assumptions about \(N\)’s value; picking \(a^{100} b^{100}\) is implicitly assuming that \(N\) is no bigger than 200 — for larger values of \(N\), \(a^{100} b^{100}\) would not be “a string whose length is at least \(N\)”.

Whatever string we pick, we have to be sure that its length is at least \(N\), no matter what number \(N\) is. So we pick, for instance, \(w = a^N b^N\). This string is in the language, and its length is at least \(N\), no matter what number \(N\) is. If we can show that this string can’t be broken down as described by the Pumping Lemma, then we’ll have shown that \(N\) doesn’t work as a threshold value, and since \(N\) was an arbitrary number, we will have shown
that there is no threshold value for $L$ and hence $L$ is not regular. So let’s show that $w = a^N b^N$ can’t be broken down appropriately.

We need to show that you can’t write $w = a^N b^N$ as $w = xyz$ where $x$ and $y$ together contain at most $N$ symbols, $y$ isn’t empty, and all the strings $xz$, $xyyz$, $xyyyz$, etc. are still in $L$, i.e. of the form $a^n b^n$ for some number $n$. The best way to do this is to show that any choice for $y$ (with $x$ being whatever precedes it and $z$ being whatever follows) that satisfies the first two requirements fails to satisfy the third. So what are our possible choices for $y$? Well, since $x$ and $y$ together can contain at most $N$ symbols, and $w$ starts with $N$ a’s, both $x$ and $y$ must be made up entirely of a’s; since $y$ can’t be empty, it must contain at least one a and (from (i)) no more than $N$ a’s. So the possible choices for $y$ are $y = a^k$ for some $1 \leq k \leq N$. We want to show now that none of these choices will satisfy the third requirement by showing that for any value of $k$, at least one of the strings $xz$, $xyyz$, $xyyyz$, etc will not be in $L$. No matter what value we try for $k$, we don’t have to look far for our rogue string: the string $xz$, which is $a^N b^N$ with $k$ a’s deleted from it, looks like $a^{N-k} b^N$, which is clearly not of the form $a^n b^n$. So the only $y$’s that satisfy (i) and (ii) don’t satisfy (iii); so $w$ can’t be broken down as required; so $N$ is not a threshold value for $L$; and since $N$ was an arbitrary number, there is no threshold value for $L$; so $L$ is not regular.

The fact that languages like $\{a^n b^n \mid n \geq 0\}$ and $\{a^p \mid p \text{ is prime}\}$ are not regular is a severe blow to any idea that regular expressions or finite-state automata capture the language-generation or language-recognition capabilities of a computer: They are both languages that we could easily write programs to recognize. It is not clear how the expressive power of regular expressions could be increased, nor how one might modify the FSA model to obtain a more powerful one. However, in the next chapter you will be introduced to the concept of a grammar as a tool for generating languages. The simplest grammars still only produce regular languages, but you will see that more complicated grammars have the power to generate languages far beyond the realm of the regular.

**Exercises**

1. Use the Pumping Lemma to show that the following languages over $\{a, b\}$ are not regular.
   a) $L_1 = \{x \mid n_a(x) = n_b(x)\}$
   b) $L_2 = \{xx \mid x \in \{a, b\}^*\}$
   c) $L_3 = \{xx^R \mid x \in \{a, b\}^*\}$
   d) $L_4 = \{a^n b^m \mid n < m\}$