CHAPTER 7

Regular Expressions

Most programmers and other power-users of computer systems have used tools that match text patterns. You may have used a Web search engine with a pattern like `travel cancun OR acapulco`*, trying to find information about either of two travel destinations. You may have used a search tool with a pattern like* `gr[ea]y`*, trying to find files with occurrences of either* `grey` *or* `gray`*. Any such pattern for text works as a definition of a formal language. Some strings match the pattern and others do not; the language defined by the pattern is the set of strings that match. In this chapter we study a particular kind of formal text pattern called a* **regular expression**. *Regular expressions have many applications. They also provide an unexpected affirmation of the importance of some of the theoretical concepts from previous chapters.*

This is a common occurrence in mathematics. The first time a young student sees the mathematical constant π*, it looks like just one more school artifact: one more arbitrary symbol whose definition to memorize for the next test. Later, if he or she persists, this perception changes. In many branches of mathematics and with many practical applications,* π *keeps on turning up. "There it is again!" says the student, thus joining the ranks of mathematicians for whom mathematics seems less like an artifact invented and more like a natural phenomenon discovered.*

So it is with regular languages. We have seen that DFAs and NFAs have equal definitional power. It turns out that regular expressions also have exactly that same definitional power: they can be used to define all the regular languages and only the regular languages. There it is again!

7.1 Regular Expressions, Formally Defined

Let's define two more basic ways of combining languages. The concatenation of two languages is the set of strings that can be formed by concatenating an element of the first language with an element of the second language:

If L_1 and L_2 are languages, the concatenation of L_1 and L_2 is $L_1L_2 = \{xy \mid x \in L_1$ and $y \in L_2\}$.

The Kleene closure of a language is the set of strings that can be formed by concatenating any number of strings, each of which is an element of that language:

If L is a language, the Kleene closure of L is $L^* = \{x_1x_2 \ldots x_n \mid n \geq 0$, with all $x_i \in L\}$.

Note that this generalizes the definition we used in Chapter 1 for the Kleene closure of an alphabet.

There is a common mistake to guard against as you read the Kleene closure definition. It does *not* say that L^* is the set of strings that are concatenations of zero or more copies of some string in L. That would be $\{x^n \mid n \geq 0$, with $x \in L\}$; compare that with the actual definition above. The actual definition says that L^* is the set of strings that are concatenations of zero or more substrings, each of which is in the language L. Each of those zero or more substrings may be a different element of L. For example, the language $\{ab, cd\}^*$ is the language of strings that are concatenations of zero of more things, each of which is either ab or cd: $\{\varepsilon, ab, cd, abab, abcd, cdab, cdcd, \ldots\}$. Note that because the definition allows *zero or more*, L^* always includes ε.

Now we can define regular expressions.

A regular expression is a string r that denotes a language $L(r)$ over some alphabet Σ. The six kinds of regular expressions and the languages they denote are as follows. First, there are three kinds of *atomic* regular expressions:

1. Any symbol $a \in \Sigma$ is a regular expression with $L(a) = \{a\}$.
2. The special symbol ε is a regular expression with $L(\varepsilon) = \{\varepsilon\}$.
3. The special symbol $\varnothing$ is a regular expression with $L(\varnothing) = \{\}$.

There are also three kinds of *compound* regular expressions, which are built from smaller regular expressions, here called r, r_1, and r_2:

4. $(r_1 + r_2)$ is a regular expression with $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
5. (r_1r_2) is a regular expression with $L(r_1r_2) = L(r_1)L(r_2)$
6. $(r)^*$ is a regular expression with $L((r)^*) = (L(r))^*$

The parentheses in compound regular expressions may be omitted, in which case * has highest precedence and + has lowest precedence.

Regular expressions make special use of the symbols ε, ∅, +, *, (, and), so for simplicity we will assume that these special symbols are not included in Σ. Regular expressions can look just like ordinary strings, so you will occasionally have to rely on the context to decide whether a string such as *abc* is just a string or a regular expression denoting the language {*abc*}.

Unfortunately, the term *regular expression* is heavily overloaded. It is used for the text patterns of many different tools like awk, sed, and grep, many languages like Perl, Python, Ruby, and PHP, and many language libraries like those for Java and the .NET languages. Each of these applications means something slightly different by *regular expression*, and we will see more about these applications later.

7.2 Examples of Regular Expressions

Using just the six basic kinds of regular expressions, you can define quite a wide variety of languages. For starters, any simple string denotes the language containing just that string:

Regular expression: ab
Language denoted: {ab}

Notice that the formal definition permits this because *a* and *b*, by themselves, are atomic regular expressions of type 1; they are combined into a compound regular expression of type 5; and the parenthesis in (*ab*) are omitted since they are unnecessary.

Any finite language can be defined, just by listing all the strings in the language, separated by the + operator:

Regular expression: *ab* + *c*
Language denoted: {*ab, c*}

Again we were able to omit all the parentheses from the fully parenthesized form ((*ab*) + *c*). The inner pair is unnecessary because + has lower precedence than concatenation. Thus, *ab* + *c* is equivalent to (*ab*) + *c*, not to *a*(*b* + *c*), which would denote a different language: {*ab*, *ac*}.

The only way to define an infinite language using regular expressions is with the Kleene star:

Regular expression: *ba**
Language denoted: $\{ba^n\} = \{b, ba, baa, baaa, ...\}$

The Kleene star has higher precedence than concatenation. *ba** is equivalent to *b*(*a**), not to (*ba*)*, which would denote the language $\{(ba)^n\}$.

$L((r)^*)$ is the language of strings that are concatenations of zero or more substrings, each of which is in the language $L(r)$. Because this is *zero or more* and not *one or more*, we always have $\varepsilon \in L((r)^*)$. Each of the concatenated substrings can be a different element of $L(r)$. For example:

Regular expression: $(a + b)^*$
Language denoted: $\{a, b\}^*$

In that example the parentheses were necessary, since * has higher precedence than +. The regular expression $a + b^*$ would denote the language $\{a\} \cup \{b^n\}$.

It is important to understand that $(a + b)^*$ is not the same as $(a^* + b^*)$. Any confusion about this point will cause big problems as we progress. In $L((a + b)^*)$ we concatenate zero or more things together, each of which may be either an a or a b. That process can construct every string over the alphabet $\{a, b\}$. In $L(a^* + b^*)$ we take the union of two sets: the set of all strings over the alphabet $\{a\}$, and the set of all strings over the alphabet $\{b\}$. That result contains all strings of only *a*s and all strings of only *b*s, but does not contain any strings that contain both *a*s and *b*s.

Occasionally the special symbol ε appears in a regular expression, when the empty string is to be included in the language:

Regular expression: $ab + \varepsilon$
Language denoted: $\{ab, \varepsilon\}$

The special symbol ∅ appears very rarely. Without it there is no other way to denote the empty set, but that is all it is good for. It is not useful in compound regular expressions, since for any regular expression r we have $L((r)\varnothing) = L(\varnothing(r)) = \{\}$, $L(r + \varnothing) = L(\varnothing + r) = L(r)$, and $L(\varnothing^*) = \{\varepsilon\}$.

The subexpressions in a compound expression may be compound expressions themselves. This way, compound expressions of arbitrary complexity may be developed:

Regular expression: $(a + b)(c + d)$
Language denoted: $\{ac, ad, bc, bd\}$

Regular expression: $(abc)^*$
Language denoted: $\{(abc)^n\} = \{\varepsilon, abc, abcabc, abcabcabc, \ldots\}$

Regular expression: a^*b^*
Language denoted: $\{a^nb^m\} = \{\varepsilon, a, b, aa, ab, bb, aaa, aab, \ldots\}$

Regular expression: $(a + b)^*aa(a + b)^*$
Language denoted: $\{x \in \{a, b\}^* \mid x \text{ contains at least two consecutive } a\text{s}\}$

Regular expression: $(a + b)^*a(a + b)^*a(a + b)^*$
Language denoted: $\{x \in \{a, b\}^* \mid x \text{ contains at least two } a\text{s}\}$

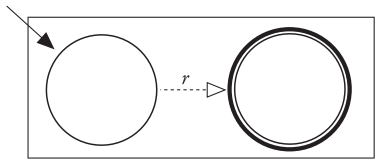
Regular expression: $(a^*b^*)^*$
Language denoted: $\{a, b\}^*$

In this last example, the regular expression $(a^*b^*)^*$ turns out to denote the same language as the simpler $(a + b)^*$. To see why, consider that $L(a^*b^*)$ contains both a and b. Those two symbols alone are enough for the Kleene star to build all of $\{a, b\}^*$. In general, whenever $\Sigma \subseteq L(r)$, then $L((r)^*) = \Sigma^*$.

7.3 For Every Regular Expression, a Regular Language

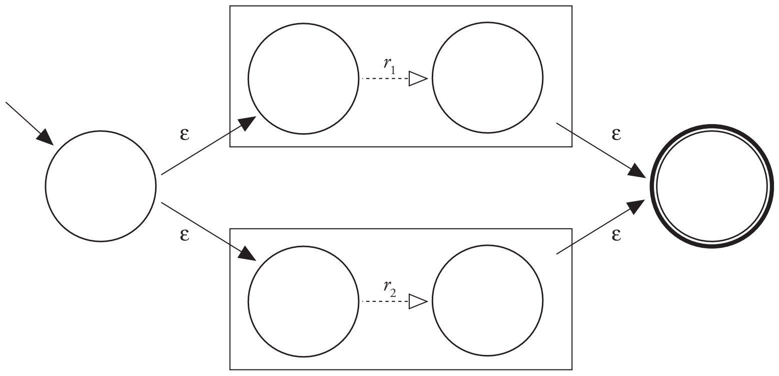
We will now demonstrate that any language defined by a regular expression can be defined by both of the other mechanisms already studied in this book: DFAs and NFAs. To prove this, we will show how to take any regular expression and construct an equivalent NFA. We choose NFAs for this, rather than DFAs, because of a property observed in Section 5.2: large NFAs can be built up as combinations of smaller ones by the judicious use of ε-transitions.

To make the constructed NFAs easy to combine with each other, we will make sure they all have exactly one accepting state, not the same as the start state. For any regular expression r we will show how to construct an NFA N with $L(N) = L(r)$ that can be pictured like this:



The constructed machine will have a start state, a single accepting state, and a collection of other states and transitions that ensure that there is a path from the start state to the accepting state if and only if the input string is in $L(r)$.

Because all our NFAs will have this form, they can be combined with each other very neatly. For example, if you have NFAs for $L(r_1)$ and $L(r_2)$, you can easily construct an NFA for $L(r_1 + r_2)$:

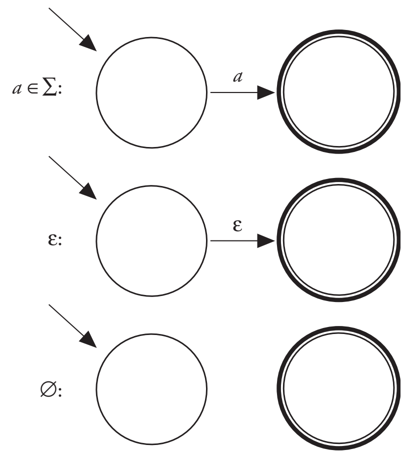


In this new machine there is a new start state with ε-transitions to the two original start states and a new accepting state with ε-transitions from the two original accepting states (which are no longer accepting states in the new machine). Clearly the new machine has a path from the start state to the accepting state if and only if the input string is in $L(r_1 + r_2)$. And it has the special form—a single accepting state, not the same as the start state—which ensures that it can be used as a building block in even larger machines.

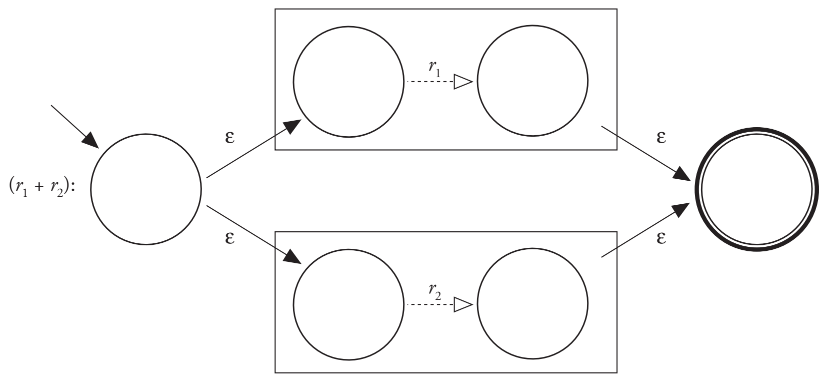
The previous example shows how to do the construction for one kind of regular expression—$r_1 + r_2$. The basic idea of the following proof sketch is to show that the same construction can be done for all six kinds of regular expressions.

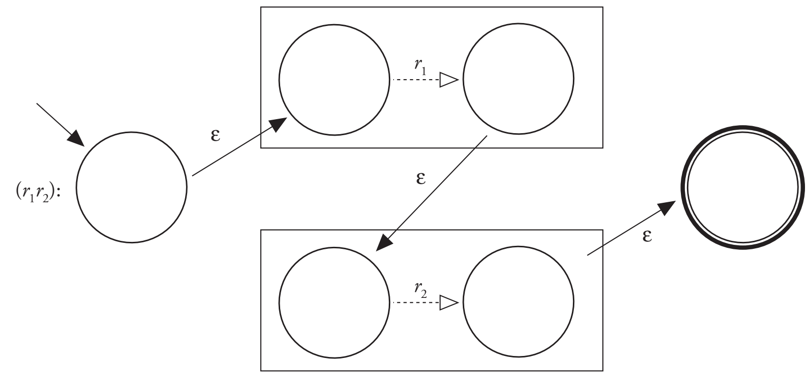
Lemma 7.1: If r is any regular expression, there is some NFA N that has a single accepting state, not the same as the start state, with $L(N) = L(r)$.

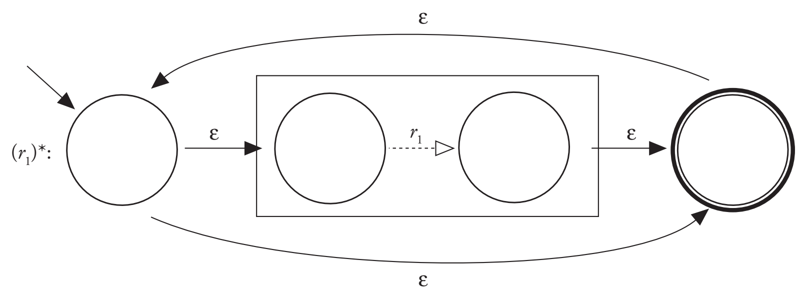
Proof sketch: For any of the three kinds of atomic regular expressions, an NFA of the desired kind can be constructed as follows:



For any of the three kinds of compound regular expressions, given appropriate NFAs for regular subexpressions r_1 and r_2, an NFA of the desired kind can be constructed as follows:







Thus, for any regular expression *r*, we can construct an equivalent NFA with a single accepting state, not the same as the start state.

7.4 Regular Expressions and Structural Induction

The proof sketch for Lemma 7.1 leaves out a number of details. A more rigorous proof would give the 5-tuple form for each of the six constructions illustrated and show that each machine actually accepts the language it is supposed to accept. More significantly, a rigorous proof would be organized as a *structural induction*.

As we have observed, there are as many different ways to prove something with induction as there are to program something with recursion. Our proofs in Chapter 3 concerning DFAs used induction on a natural number—the length of the input string. That is the style of induction that works most naturally for DFAs. Structural induction performs induction on a recursively defined structure and is the style of induction that works most naturally for regular expressions. The base cases are the atomic regular expressions, and the inductive cases are the compound forms. The inductive hypothesis is the assumption that the proof has been done for structurally simpler cases; for a compound regular expression *r*, the inductive hypothesis is the assumption that the proof has been done for *r*'s subexpressions.

Using structural induction, a more formal proof of Lemma 7.1 would be organized like this:

Proof: By induction on the structure of *r*.

Base cases: When *r* is an atomic expression, it has one of these three forms:

1. Any symbol $a \in \Sigma$ is a regular expression with $L(a) = \{a\}$.
2. The special symbol ε is a regular expression with $L(\varepsilon) = \{\varepsilon\}$.
3. The special symbol $\varnothing$ is a regular expression with $L(\varnothing) = \{\}$.

(For each atomic form you would give the NFA, as in the previous proof sketch.)

Inductive cases: When *r* is a compound expression, it has one of these three forms:

4. $(r_1 + r_2)$ is a regular expression with $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
5. (r_1r_2) is a regular expression with $L(r_1r_2) = \{xy \mid x \in L(r_1) \text{ and } y \in L(r_2)\}$
6. $(r_1)^*$ is a regular expression with
$$L((r_1)^*) = \{x_1x_2 \dots x_n \mid n \geq 0, \text{ with all } x_i \in L(r_1)\}$$

By the inductive hypothesis, there is an NFA that has a single accepting state, not the same as the start state, for r_1 and r_2.

(For each compound form you would give the ε-NFA, as in the previous proof sketch, and show that it accepts the desired language.)

A proof using this style of induction demonstrates that Lemma 7.1 holds for any atomic regular expression and then shows that whenever it holds for some expressions r_1 and r_2, it also holds for $(r_1 + r_2)$, (r_1r_2), and $(r_1)^*$. It follows that the lemma holds for all regular expressions.

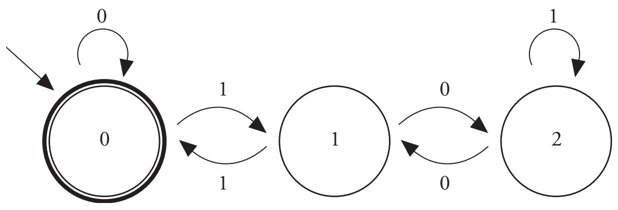
7.5 For Every Regular Language, a Regular Expression

There is a way to take any NFA and construct an equivalent regular expression.

> **Lemma 7.2:** If N is any NFA, there is some regular expression r with $L(N) = L(r)$.

The construction that proves this lemma is rather tricky, so it is relegated to Appendix A. For now, a short example will suffice.

Consider again this NFA (which is also a DFA) from Chapter 3:



We proved that this machine accepts the language of strings that are binary representations of numbers that are divisible by three. Now we will construct a regular expression for this same language. It looks like a hard problem; the trick is to break it into easy pieces.

When you see an NFA, you normally think only of the language of strings that take it from the start state to end in any accepting state. But let's consider some other languages that are relevant to this NFA. For example, what is the language of strings that take the machine from state 2 back to state 2, any number of times, *without passing through states 0 or 1*? Any string of zero or more 1s would do it, and that is an easy language to give a regular expression for:

$$1^*$$

Next, a bigger piece: what is the language of strings that take the machine from 1 back to 1, any number of times, *without passing through state 0*? The machine has no transition that allows it to go from 1 directly back to 1. So each single trip from 1 back to 1 must follow these steps:

1. Go from 1 to 2.
2. Go from 2 back to 2, any number of times, without passing through 0 or 1.
3. Go from 2 to 1.

The first step requires a 0 in the input string. The second step is a piece we already have a regular expression for: 1*. The third step requires a 0 in the input string. So the whole language, the language of strings that make the machine do those three steps repeated any number of times, is

(01*0)*

Next, a bigger piece: what is the language of strings that take the machine from 0 back to 0, any number of times? There are two ways to make a single trip from 0 back to 0. The machine can make the direct transition from state 0 to state 0 on an input symbol 0 or it can follow these steps:

1. Go from 0 to 1.
2. Go from 1 back to 1, any number of times, without passing through 0.
3. Go from 1 to 0.

The first step requires a 1 in the input string. The second step is a piece we already have a regular expression for: (01*0)*. The third step requires a 1 in the input string. So the whole language, the language of strings that make the machine go from state 0 back to state 0 any number of times, is

(0 + 1(01*0)*1)*

This also defines the whole language of strings accepted by the NFA—the language of strings that are binary representations of numbers that are divisible by three.

The proof of Lemma 7.2 is a construction that builds a regular expression for the language accepted by any NFA. It works something like the example just shown, defining the language in terms of smaller languages that correspond to restricted paths through the NFA. Appendix A provides the full construction. But before we get lost in those details, let's put all these results together:

Theorem 7.1 (Kleene's Theorem): A language is regular if and only if it is $L(r)$ for some regular expression r.

Proof: Follows immediately from Lemmas 7.1 and 7.2.

DFAs, NFAs, and regular expressions all have equal power for defining languages.

Exercises

EXERCISE 1

Give a regular expression for each of the following languages.

a. $\{abc\}$
b. $\{abc, xyz\}$
c. $\{a, b, c\}^*$
d. $\{ax \mid x \in \{a, b\}^*\}$
e. $\{axb \mid x \in \{a, b\}^*\}$
f. $\{(ab)^n\}$
g. $\{x \in \{a, b\}^* \mid x$ contains at least three consecutive as$\}$
h. $\{x \in \{a, b\}^* \mid$ the substring bab occurs somewhere in $x\}$
i. $\{x \in \{a, b\}^* \mid x$ starts with at least three consecutive as$\}$
j. $\{x \in \{a, b\}^* \mid x$ ends with at least three consecutive as$\}$
k. $\{x \in \{a, b\}^* \mid x$ contains at least three as$\}$
l. $\{x \in \{0, 1\}^* \mid x$ ends in either 0001 or 1000$\}$
m. $\{x \in \{0, 1\}^* \mid x$ either starts with 000 or ends with 000, or both$\}$
n. $\{a^n \mid n$ is even$\} \cup \{b^n \mid n$ is odd$\}$
o. $\{(ab)^n\} \cup \{(aba)^n\}$
p. $\{x \in \{a\}^* \mid$ the number of as in x is odd$\}$
q. $\{x \in \{a, b\}^* \mid$ the number of as in x is odd$\}$
r. $\{x \in \{a, b\}^* \mid$ the number of as in x is odd$\} \cup \{b^n \mid n$ is odd$\} \cup \{(aba)^n\}$
s. $\{a^n \mid n$ is divisible by at least one of the numbers 2, 3, or 5$\}$
t. $\{x \in \{a, b\}^* \mid x$ has no two consecutive as$\}$
u. $\{xy \in \{a, b\}^* \mid$ the number of as in x is odd and the number of bs in y is even$\}$

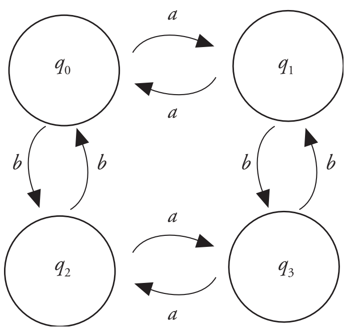
EXERCISE 2

For each of these regular expressions, give two NFAs: the exact one constructed by the proof of Lemma 7.1, and the smallest one you can think of.

a. $\varnothing$
b. ε
c. a
d. $0 + 1$
e. $(00)^*$
f. ab^*

EXERCISE 3

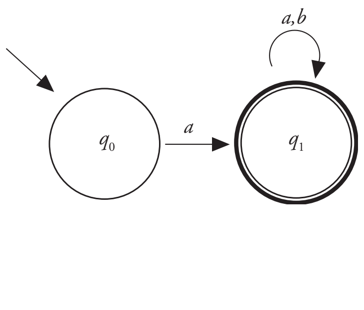
For the following DFA, give a regular expression for each of the languages indicated. When the question refers to a machine "passing through" a given state, that means entering and then exiting the state. Merely starting in a state or ending in it does not count as "passing through."



a. the language of strings that make the machine, if started in q_0, end in q_0, *without passing through* q_0, q_1, q_2, or q_3
b. the language of strings that make the machine, if started in q_0, end in q_2, *without passing through* q_0, q_1, q_2, or q_3
c. the language of strings that make the machine, if started in q_2, end in q_0, *without passing through* q_0, q_1, q_2, or q_3
d. the language of strings that make the machine, if started in q_2, end in q_2, *without passing through* q_0 or q_1
e. the language of strings that make the machine, if started in q_2, end in q_0, *without passing through* q_0 or q_1
f. the language of strings that make the machine, if started in q_0, end in q_2, *without passing through* q_0 or q_1
g. the language of strings that make the machine, if started in q_1, end in q_1, *without passing through* q_0 or q_1
h. the language of strings that make the machine, if started in q_1, end in q_2, *without passing through* q_0 or q_1
i. the language of strings that make the machine, if started in q_0, end in q_0, *without passing through* q_0 or q_1
j. the language of strings that make the machine, if started in q_1, end in q_1, *without passing through* q_0

EXERCISE 4

(This exercise refers to material from Appendix A.) Let M be this NFA:

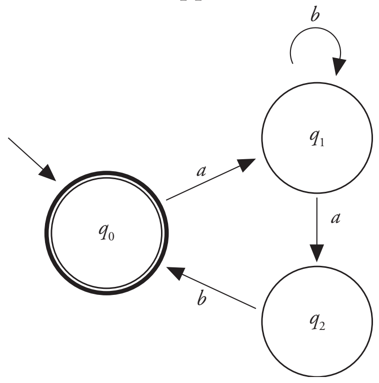


Give the simplest regular expression you can think of for each of the following internal languages of M.

a. $L(M, 0, 0, 2)$
b. $L(M, 0, 0, 1)$
c. $L(M, 0, 0, 0)$
d. $L(M, 0, 1, 2)$
e. $L(M, 0, 1, 1)$
f. $L(M, 0, 1, 0)$
g. $L(M, 1, 1, 2)$
h. $L(M, 1, 1, 1)$
i. $L(M, 1, 1, 0)$
j. $L(M, 1, 0, 2)$
j. $L(M, 1, 0, 1)$
l. $L(M, 1, 0, 0)$

EXERCISE 5

(This exercise refers to material from Appendix A.) Let M be this NFA:

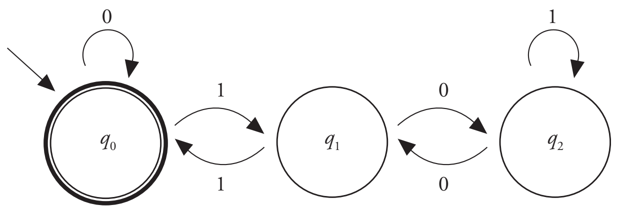


Give the simplest regular expression you can think of for each of the following internal languages of M. (Use the d2r construction if necessary.)

a. $L(M,2,2,3)$
b. $L(M,2,2,2)$
c. $L(M,1,1,3)$
d. $L(M,1,1,2)$
e. $L(M,1,1,1)$
f. $L(M,0,0,3)$
g. $L(M,0,0,2)$
h. $L(M,0,0,1)$
i. $L(M,0,0,0)$

EXERCISE 6

(This exercise refers to material from Appendix A.) Let *M* be this DFA:



Give the simplest regular expression you can think of for each of the following internal languages of *M*. (Use the d2r construction if necessary.)

a. $L(M, 2, 2, 3)$
b. $L(M, 2, 2, 2)$
c. $L(M, 1, 1, 3)$
d. $L(M, 1, 1, 2)$
e. $L(M, 1, 1, 1)$
f. $L(M, 0, 0, 3)$
g. $L(M, 0, 0, 2)$
h. $L(M, 0, 0, 1)$
i. $L(M, 0, 0, 0)$

EXERCISE 7

(This exercise refers to material from Appendix A.) The following code is used to verify the regular expression computed by d2r for the example in Section A.3:

```
public static void main(String args[]) {
    NFA N = new NFA();
    State q0 = N.getStartState();
    State q1 = N.addState();
    N.addX(q0,'a',q0);
    N.addX(q0,'b',q1);
    N.addX(q1,'b',q1);
    N.addX(q1,'a',q0);
    N.makeAccepting(q1);
    System.out.println("final:" + N.d2r(0,1,0));
}
```

This code makes use of two classes, `State` and `NFA`. The `NFA` class represents an NFA. When an `NFA` object is constructed, it initially contains only one state, the start state, with no accepting states and no transitions. Additional states are added dynamically by calling the `addState` method, which returns an object of the `State` class. Each transition on a symbol is added dynamically by calling the `addX` method, giving the source state, the symbol, and the target state. States are labeled as accepting by calling the `makeAccepting` method. Finally, the d2r construction is performed by calling the `d2r` method, passing it the integers *i*, *j*, and *k*. (States are considered to be numbered in the order they were added to the `NFA`, with the start state at number 0.) The `d2r` method returns a `String`. The code shown above creates an `NFA` object, initializes it as the example NFA, and prints out the regular expression for the language it accepts.

Implement the `State` and `NFA` classes according to this specification, along with any auxiliary classes you need. You need not implement ε-transitions in the NFA, and you may represent ε in regular expressions using the symbol `e`. Test your classes with the test code shown above, and check that your `d2r` returns the correct regular expression. (Recall that, for readability, some of the parentheses in the regular expression in Section A.3 were omitted. Your result may include more parentheses than shown there.)

EXERCISE 8

For any string x, define x^R to be that same string reversed. The intuition is plain enough, but to support proofs we'll need a formal definition:

$\varepsilon^R = \varepsilon$
$(ax)^R = x^R a$, for any symbol a and string x

We'll use the same notation to denote the set formed by reversing every string in a language:

$A^R = \{x^R \mid x \in A\}$

Using these definitions, prove the following properties of reversal:

a. Prove that for any strings x and y, $(xy)^R = y^R x^R$. *Hint:* Use induction on $|x|$.
b. Prove that for any string x, $(x^R)^R = x$. *Hint:* Use induction on $|x|$ and the result from Part a.
c. Prove that for any languages A and B, $(A \cup B)^R = A^R \cup B^R$.
d. Prove that for any languages A and B, $(AB)^R = B^R A^R$. You'll need the result from Part a.
e. Prove that for any language A, $(A^*)^R = (A^R)^*$. You'll need the result from Part a.
f. Prove that the regular languages are closed for reversal. *Hint:* Using structural induction, show that for every regular expression r, there is a regular expression r' with $L(r') = (L(r))^R$.