Finite Automata

Mark V. Lawson
Heriot-Watt University, Edinburgh

November 4, 2009

Contents

Preface

The theory of finite automata is the mathematical theory of a simple class of algorithms that are important in mathematics and computer science. Three papers laid the foundations of finite automata theory: Turing's 1936 paper [49] in which algorithmic problems are defined as those which can be solved by mechanical means in terms of what are now known as *Turing machines*; McCulloch and Pitts' paper [25] of 1943 in which a mathematical model of brain neurons was constructed; and Kleene's paper [20], developing his RAND report of 1951, in which McCulloch and Pitts' model was subjected to a detailed mathematical makeover resulting in the finite automata of this course.[1]

Applications of finite automata and their languages are legion:

- The book by Petzold [36] is an elementary introduction to circuit design.

- Aho, Sethi, and Ullman [1] explain how finite automata form one of the ingredients in designing compilers.

- Friedl [11] describes the thousand-and-one uses of regular expressions to professional programmers — such expressions are equivalent to finite automata as we shall prove in Chapter 5.

[1]If you want to know more about the history of finite automata, the essay by Perrin [35] is interesting, and there are surveys of important papers in Brauer [4]. The collections of papers that appear in [43] and [31] convey something of the flavour of the early work. References to work on automata theory in the former Soviet bloc can be found in [12] and [13] as well as Brauer [4].

The theory of finite automata is an established part of theoretical computer science, and so any book dealing with this subject will contain accounts of finite automata to a greater or lesser extent. Textbooks that contain chapters on finite automata, at approximately the same level as this course, are [5], [8], [18], [21], [22], [40], and [46].

- Searching for patterns in texts can be carried out efficiently using automata [9].
- The collection of papers to be found in [39] demonstrates the usefulness of finite automata in natural language processing.
- Lind and Marcus [23] show how finite automata, under the alias of 'sofic system,' can be used in encoding information, a further useful introduction to these ideas is [3].
- von Haeseler [15] uses finite automata to generate sequences of numbers.
- Sims [45] uses finite automata to describe some algorithms in group theory.
- Epstein et al [10] explain how finite automata form an important tool in combinatorial group theory and geometry.
- Thurston [48] interweaves groups, tilings, dynamical systems, and finite automata; Grigorchuk et al [14] actually build groups from automata.
- Pin [37] develops the algebraic theory of recognisable languages within finite semigroup theory.

This course is based around two main theorems: Kleene's Theorem, proved in Chapter 5, and the theorem that states that two reduced accessible automata recognising the same language are isomorphic, proved in Chapter 6. Kleene's Theorem is the first main theorem of theoretical computer science, whereas the second theorem can be used to show that every recognisable language is accepted by an essentially unique minimal automaton.

Chapter 1

Introduction to finite automata

The theory of finite automata is the mathematical theory of a simple class of algorithms that are important in mathematics and computer science. In this chapter, we set the scene for the entire course by explaining what we mean by a finite automaton and the language recognised by a finite automaton. In order to define languages, we have first to define alphabets and strings. One of the goals of this chapter is to explain why the notion of 'language' is an important one.

1.1 Alphabets and strings

Most people today are familiar with the idea of *digitising information*; that is, converting information from an analogue or continuous form to a discrete form. It is well-known that computers deal only in 0's and 1's, but users of computers do not have to communicate with them in binary; they can interact with the computer in a great variety of ways. For example, voice recognition technology enables us to input data without using the keyboard, whereas computer graphics can present output in the form of animation. But these things are only possible because of the underlying sequences of 0's and 1's that encode this information. We begin this section therefore by examining sequences of symbols and their properties.

Information in all its forms is usually represented as sequences of symbols drawn from some fixed repertoire of symbols. More formally, any set of symbols A that is used in this way is called an *alphabet*, and any finite sequence whose components are drawn from A is called a *string over A* or simply a *string*.[1] We call the elements of an alphabet *symbols* or *letters*. The number of symbols in an alphabet A is denoted by $|A|$. The alphabets in this course will always be finite.

Examples 1.1.1 Here are a few examples of alphabets you may have encountered.

(1) An alphabet suitable for describing the detailed workings of a computer is $\{0, 1\}$.

(2) An alphabet for representing natural numbers in base 10 is

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

This alphabet is one of the great human inventions.

(3) An alphabet suitable for writing stories in English is

$$\{a, \ldots, z, A \ldots, Z, ?, \ldots\},$$

upper and lower case letters together with punctuation symbols and a space symbol to separate different words. Alphabets, whether Latin, Greek, Hebrew etc, are another great human invention.

(4) An alphabet for formal logic is $\{\exists, \forall, \neg, \wedge, \ldots\}$. This alphabet is important in writing mathematics.

(5) The alphabet used in describing a programming language is called the set of *tokens* of the language. For example, in the C language, the following are all tokens:

main,   printf,   {,   }.

(6) DNA is constructed from four main types of molecules: adenine (A), cytosine (C), guanine (G), and thymine (T). Sequences of these molecules, and so strings over the alphabet $\{A, C, G, T\}$, form the basis of genes.

□

[1]The term *word* is often used instead of string.

The symbols in an alphabet do not have to be especially simple. An alphabet could consist of pictures, or each element of an alphabet could itself be a sequence of symbols. Thus the set of all Chinese characters is an alphabet in our sense although it is not an alphabet in the linguistic sense, as is the set of all words in an ordinary dictionary — a word like 'egalitarianism' would, in this context, be regarded as a single symbol. An important example of using sequences of symbols over one alphabet to represent the elements of another alphabet occurs with ASCII encoding, and also forms the basis of data-compression and error-correction codes. You might wonder why, when all information can be encoded in binary, we do not just stick with the alphabet $\{0, 1\}$. The reason is one of convenience: binary is good for computers and bad for people. That said, most of the alphabets we use in this course will just have a few elements but, again, that is just for convenience.

A string is a list and so it is formally written using brackets and commas to separate components. Thus $(0, 1, 1, 1, 0)$ is a string over the alphabet $A = \{0, 1\}$, whereas $(\mathrm{to}, \mathrm{be}, \mathrm{or}, \mathrm{not}, \mathrm{to}, \mathrm{be})$ is a string over the alphabet whose elements are the words in an English dictionary. The string $()$ is the empty string. However, for the remainder of this course, we shall write strings without brackets and commas and so for instance we write 01110 rather than $(0, 1, 1, 1, 0)$. The empty string needs to be recorded in some way and we denote it by ε. The set of all strings over the alphabet A is denoted by A^*, read A *star*, and the set of all strings except the empty one is denoted by A^+, read A *plus*.

Two strings u and v over an alphabet A are *equal* if they contain the same symbols in the same order. More formally, $x = y$ iff either $x = y = \varepsilon$ or $|x| = |y| = n > 0$ and for $1 \leq i \leq n$ we have that $x_i = y_i$.

Given two strings $x, y \in A^*$, we can form a new string $x \cdot y$, called the *concatenation of x and y*, by simply adjoining the symbols in y to those in x. For example, if $A = \{0, 1\}$ then both 0101 and 101010 are strings over A. The concatenation of 0101 and 101010 is denoted $0101 \cdot 101010$ and is equal to the string 0101101010. We shall usually denote the concatenation of x and y by xy rather than $x \cdot y$. The string ε has a special property with respect to concatenation: for each string $x \in A^*$ we clearly have that $\varepsilon x = x = x\varepsilon$.

There is one point that needs to be emphasised: **the order in which strings are concatenated is important**. For example, if $A = \{a, b\}$ and $u = ab$ and $v = ba$ then $uv = abba$ and $vu = baab$ and clearly $uv \neq vu$. We have all been made painfully familiar with this fact: the spelling of the word 'concieve' is wrong, whereas the spelling 'conceive' is correct. This is because 'order matters' in spelling. In the case where A consists of only one letter, then the order in which we concatenate strings is immaterial. For example, if $A = \{a\}$ then strings in A^* are just sequences of a's, and clearly, the order in which we concatenate strings of a's is not important.

Given three strings x, y, and z, there are two distinct ways to concatenate them in this order: we can concatenate x and y first to obtain xy and then concatenate xy with z to obtain xyz, or we can concatenate y and z first to obtain yz and then concatenate x with yz to obtain xyz again. In other words, $(xy)z = x(yz)$. We say that concatenation is *associative.*

Remark A set S equipped with an associative binary operation is called a *semigroup.* Depending on the set S, we might use other symbols to denote the binary operation: $+$, $\times$, $\circ$, $*$ etc. When proving results about arbitrary semigroups we usually use concatenation to denote the binary operation. An *identity element* in the semigroup S is an element e such that $es = s = se$ for all $s \in S$. It can be easily proved that if a semigroup has an identity then it has exactly one. A semigroup with an identity is called a *monoid.* It follows that A^* is a monoid with respect to the binary operation of concatenation and with identity the empty string. This monoid is called the *free monoid (on the set A).* Clearly A^+ is just a semigroup and it is called the *free semigroup (on the set A).*

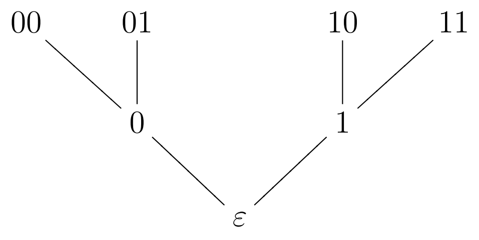
If x is a string then we write x^n, when $n \geq 1$, to mean the concatenation of x with itself n-times. We define $x^0 = \varepsilon$. For example, if $x = ba$ then $(ba)^2 = baba$. The usual laws of indices hold: if $m, n \geq 0$ then $x^m x^n = x^{m+n}$.

If w is a string then $|\, w \,|$ denotes the total number of symbols appearing in w and is called the *length of* w. If $a \in A$ then $|\, w \,|_a$ is the total number of a's appearing in w. For example, $|\, \varepsilon \,| = 0$, and $|\, 01101 \,| = 5$; $|\, 01101 \,|_0 = 2$, and $|\, 01101 \,|_1 = 3$. If $x, y \in A^*$ then $|\, xy \,| = |\, x \,| + |\, y \,|$; when we concatenate two strings the length of the result is the sum of the lengths of the two strings.

When discussing strings over an alphabet, it is useful to have a standard way of listing them. This can easily be done using what is known as the *tree order*[2] on A^*. Let $A = \{a_1, \ldots, a_n\}$ be an alphabet. Choose a fixed linear order for the elements of A. This is usually obvious, for example, if $A = \{0, 1\}$ then we would assume that $0 < 1$ but in principle any ordering of

[2]Also known as the 'length-plus-lexicographic order,' which is more of a mouthful, and the 'ShortLex order.'

the elements of the alphabet may be chosen, but if a non-standard ordering is to be used then it has to be explicitly described. We now grow a tree, called the *tree over* A^*, whose root is ε and whose vertices are labelled with the elements of A^* according to the following recipe: if w is a vertex, then the vertices growing out of w are $wa_1, \ldots, wa_n$. The *tree order on* A^* is now obtained as follows: $x < y$ if and only if $|x| < |y|$, or $|x| = |y|$ and the string x occurs to the left of the string y in the tree over A^*. To make this clearer, we do a simple example. Let $A = \{0, 1\}$, where we assume $0 < 1$. The first few levels of the tree over A^* are:



Thus the tree order for A^* begins as follows

$$\varepsilon < 0 < 1 < 00 < 01 < 10 < 11 < \ldots.$$

This ordering amounts to saying that a string precedes all strictly longer strings, while all the strings of the same length are listed *lexicographically*, that is to say the way they are listed in a dictionary[3] based on the ordering of the alphabet being used.

Let $x, y, z \in A^*$. If $u = xyz$ then y is called a *factor* of u, x is called a *prefix* of u, and z is called a *suffix* of u. We call the factor y *proper* if at least one of x and z is not just the empty string. In a similar fashion we say that the prefix x (resp. suffix z) is *proper* if $x \neq u$ (resp. $z \neq u$). We say that the string u is a *substring* of the string v if $u = a_1 \ldots a_n$, where $a_i \in A$, and there exist strings $x_0, \ldots, x_n$ such that $v = x_0 a_1 x_1 \ldots x_{n-1} a_n x_n$. Let $x \in A^*$. We call a representation $x = u_1 \ldots u_n$, where each $u_i \in A^*$, a *factorisation of* x.

Example 1.1.2 Consider the string $u = abab$ over the alphabet $\{a, b\}$. Then the prefixes of u are: $\varepsilon, a, ab, aba, abab$; the suffixes of u are: $\varepsilon, b, ab, bab, abab$; and the factors of u are: $\varepsilon, a, b, ab, ba, aba, bab, abab$. The strings aa, bb, abb are examples of substrings of u. Finally, $u = ab \cdot ab$ is a factorisation of u; observe that I use the $\cdot$ to emphasise the factorisation.

[3]Also known as a *lexicon*.

Exercises 1.1

1. Write down the set of prefixes, the set of suffixes, and the set of factors of the string,

   aardvark,

   over the alphabet $\{a, \ldots, z\}$. When writing down the set of factors, list them in order of length. Find three substrings that are not factors.

2. Let $A = \{a, b\}$ with the order $a < b$. Draw the tree over A^* up to and including all strings of length 3. Arrange these strings according to the tree order.

3. Let A be an alphabet. Prove that A^* is *cancellative* with respect to concatenation, meaning that if $x, y, z \in A^*$ then $xz = yz$ implies $x = y$, and $zx = zy$ implies $x = y$.

4. Let $x, y, u, v \in A^*$. Suppose that $xy = uv$. Prove the following hold:

   (i) If $|x| > |u|$, then there exists a non-empty string w such that $x = uw$ and $v = wy$.

   (ii) If $|x| = |u|$, then $x = u$ and $y = v$.

   (iii) If $|x| < |u|$, then there exists a non-empty string w such that $u = xw$ and $y = wv$.

5. In general, if $u, v \in A^+$, then the strings uv and vu are different as we have noted. This raises the question of finding conditions under which $uv = vu$. Prove that the following two conditions are equivalent:

   (i) $uv = vu$.

   (ii) There exists a string z such that $u = z^p$ and $v = z^q$ for some natural numbers $p, q > 0$.

   You can use Question 4 in solving this problem. Proving results about strings is often no easy matter. More combinatorial properties of strings are described in [24].

6. Prove that in a semigroup there is at most one identity.

7. Determine which of the following are semigroups and which are not, and give reasons.

   (i) The set $T(X)$ of all functions defined from the set X to itself equipped with the binary operation of composition of functions.

   (ii) The set $M_n(\mathbb{R})$ of all $n \times n$ real matrices equipped with matrix multiplication.

   (iii) The set of all three dimensional vectors equipped with the vector product.

1.2 Languages

Before defining the word 'language' formally, here is a motivating example.

Example 1.2.1 Let A be the alphabet that consists of all words in an English dictionary. So A contains a very large number of elements: of the order of half a million. As we explained in Section 1.1, we can think of each English word as being a single symbol. The set A^* consists of all possible finite sequences of words. An important subset L of A^* consists of all sequences of words that form grammatically correct English sentences. Thus the sequence (to,be,or,not,to,be)$\in L$ whereas (be,be,to,to,or,not) $\notin L$. Someone who wants to understand English has to learn the rules for deciding when a string of words belongs to the set L. We can therefore think of L as being the 'English language.'[4] $\square$

This example motivates the following definition. For *any* alphabet A, *any* subset of A^* is called an *A-language*, or a *language over A* or simply a *language*.

Examples 1.2.2 Here are some examples of languages.

(1) In elementary arithmetic we use the alphabet,

$$A = \{0, \ldots, 9\} \cup \{+, \times, -, \div, =\} \cup \{(,)\}.$$

We can form the language L of all correct sums: thus the sequence $2 + 2 = 4$ is in L whereas the sequence $1 \div 0 = 42$ is not. Any totally meaningless string such as $\div + = 98 =$ also fails to be in L.

[4]In reality, membership of this set is sometimes problematic, but the languages we meet in practice will be formal and always clearly defined.

(2) In computer science, the set of all syntactically correct programs in a given computer language, such as Java, constitutes a language.

(3) Both $\emptyset$ and A^* are languages over A: the smallest and the largest, respectively.

□

We have seen that languages arise as natural languages, and as computer languages, but perhaps the most important class of languages in mathematics are those that arise from the theory of algorithms. Problems come in all shapes and sizes. For example, the problem of finding the prime factors of a natural number involves input, in the form of a natural number n, and output let us say the nondecreasing sequence of prime factors of n. For example, if we input 12 the output will be $2, 2, 3$. Some problems give much more restricted outputs. For example, the problem 'is $n \geq 2$ a prime' outputs 'yes', if n is a prime, and 'no', if n is composite. Problems whose outputs are either 'yes' or 'no' are a special class of problem called *decision problems*. Decision problems might seem a little feeble compared with problems that deliver honest-to-goodness outputs, but in fact they are very useful for two reasons. First, decision problems can be used to generate output: for example, to find the prime factors of a number n I can use a sequence of decision problems such as 'is n divisible by 2?', 'is n divisible by 3?' and so on. Thus information about decision problems can be used to get information about problems which are not themselves decision problems. Second, decision problems are much easier to handle mathematically; this is because decision problems are really languages in disguise. The reason is that the inputs to a decision problem can be coded as strings. Those strings whose corresponding inputs give the answer 'yes' then form the language corresponding to the decision problem. Thus languages can be viewed as decision problems wearing a false beard, and decision problems are important.

But we are not just interested in problems, we are much more interested in methods, or better algorithms, for solving problems. Informally, an algorithm is a step-by-step procedure for solving a problem: programs are good examples of algorithms. Thus the decision problem 'is $n \geq 2$ a prime' can be solved by the algorithm: try dividing n by each $m \leq \sqrt{n}$; if none of them works then n is prime, and if one of them does work then n is not prime. An important goal of mathematics is to try to find good algorithms for solving decision problems, which translates into trying to find good algorithms for determining whether a string belongs to a language. As we shall see, those languages which can be accepted by a finite automaton admit a very good algorithm for deciding whether a string belongs to them or not: namely, any finite automaton that recognises the language. We shall also find that not all languages can have their membership problem decided by an automaton.

1.3 Language operations

In Section 1.2, we introduced languages as they will be understood in this course. We shall now define various operations on languages: that is, ways of combining languages to make new ones.

If X is any set, then $\mathsf{P}(X)$ is the set of all subsets of X, the *power set of* X. Now let A be an alphabet. A language over A is any subset of A^*, so that the set of all languages over A is just $\mathsf{P}(A^*)$. If L and M are languages over A so are $L \cap M$, $L \cup M$ and $L \setminus M$ ('relative complement'). If L is a language over A, then $L' = A^* \setminus L$ is a language called the *complement of* L. The operations of intersection, union, and complementation are called *Boolean operations* and come from set theory. Recall that '$x \in L \cup M$' means '$x \in L$ or $x \in M$ or both.' In automata theory, we usually write $L + M$ rather than $L \cup M$ when dealing with languages.

Notation If L_i is a family of languages where $1 \leq i \leq n$, then their union will be written $\sum_{i=1}^{n} L_i$.

There are two further operations on languages that are peculiar to automata theory and extremely important: the product and the Kleene star.

Let L and M be languages. Then

$$L \cdot M = \{xy\colon x \in L \text{ and } y \in M\}$$

is called the *product of* L *and* M. We usually write LM rather than $L \cdot M$. A string belongs to LM if it can be written as a string in L *followed by* a string in M. In other words, the product operation enables us to talk about the order in which symbols or strings occur.

Examples 1.3.1 Here are some examples of products of languages.

(1) $\emptyset L = \emptyset = L\emptyset$ for any language L.

(2) $\{\varepsilon\}L = L = L\{\varepsilon\}$ for any language L.

(3) Let $L = \{aa, bb\}$ and $M = \{aa, ab, ba, bb\}$. Then

$$LM = \{aaaa, aaab, aaba, aabb, bbaa, bbab, bbba, bbbb\}$$

and

$$ML = \{aaaa, aabb, abaa, abbb, baaa, babb, bbaa, bbbb\}.$$

In particular, $LM \neq ML$ in general.

□

For a language L we define $L^0 = \{\varepsilon\}$ and $L^{n+1} = L^n \cdot L$. For $n > 0$ the language L^n consists of all strings u of the form $u = x_1 \dots x_n$ where $x_i \in L$.

The *Kleene star* of a language L, denoted L^*, is defined to be

$$L^* = L^0 + L^1 + L^2 + \dots .$$

We also define

$$L^+ = L^1 + L^2 + \dots .$$

Examples 1.3.2 Here are some examples of the Kleene star of languages.

(1) $\emptyset^* = \{\varepsilon\}$ and $\{\varepsilon\}^* = \{\varepsilon\}$.

(2) The language $\{a^2\}^*$ consists of the strings,

$$\varepsilon, a^2, a^4 = a^2a^2, a^6 = a^2a^2a^2, \dots .$$

In other words, all strings over the alphabet $\{a\}$ of even length (**remember**: the empty string has even length because 0 is an even number).

(3) A string u belongs to $\{ab, ba\}^*$ if it is empty or if u can be factorised $u = x_1 \dots x_n$ where each x_i is either ab or ba. Thus the string $abbaba$ belongs to the language because $abbaba = ab \cdot ba \cdot ba$, but the string $abaaba$ does not because $abaaba = ab \cdot aa \cdot ba$.

□

Notation We can use the Boolean operations, the product, and the Kleene star to describe languages. For example, $L = \{a, b\}^* \setminus \{a, b\}^*\{aa, bb\}\{a, b\}^*$ consists of all strings over the alphabet $\{a, b\}$ that do not contain a doubled symbol. Thus the string *ababab* is in L whereas *abaaba* is not. When languages are described in this way, it quickly becomes tedious to keep having to write down the brackets { and }. Consequently, from now on we shall omit them. If brackets are needed to avoid ambiguity we use (and). This notation is made rigorous in Section 5.1.

Examples 1.3.3 Here are some examples of languages over the alphabet $A = \{a, b\}$ described using our notational convention above.

(1) We can write A^* as $(a + b)^*$. To see why, observe that

$$A^* = \{a, b\}^* = (\{a\} + \{b\})^* = (a + b)^*,$$

where the last equality follows by our convention above. We have to insert brackets because $a + b^*$ is a different language. See Exercises 1.3.

(2) The language $(a + b)^3$ consists of all 8 strings of length 3 over A. This is because $(a + b)^3$ means $(a + b)(a + b)(a + b)$. A string x belongs to this language if we can write it as $x = a_1a_2a_3$ where $a_1, a_2, a_3 \in \{a, b\}$.

(3) The language $aab(a+b)^*$ consists of all strings that begin with the string aab, whereas the language $(a+b)^*aab$ consists of all strings that end in the string aab. The language $(a + b)^*aab(a + b)^*$ consists of all strings that contain the string aab as a factor.

(4) The language $(a + b)^*a(a + b)^*a(a + b)^*b(a + b)^*$ consists of all strings that contain the string aab as a substring.

(5) The language $aa(a + b)^* + bb(a + b)^*$ consists of all strings that begin with a double letter.

(6) The language $(aa + ab + ba + bb)^*$ consists of all strings of even length.

□

REMEMBER! The symbol $+$ means *or*, whereas the symbol $\cdot$ means *followed by*. If you muddle them up, you will get the wrong answer.

Exercises 1.3

1. Let $L = \{ab, ba\}$, $M = \{aa, ab\}$ and $N = \{a, b\}$. Write down the following.

   (i) LM.

   (ii) LN.

   (iii) $LM + LN$.

   (iv) $M + N$.

   (v) $L(M + N)$.

   (vi) $(LM)N$.

   (vii) MN.

   (viii) $L(MN)$.

2. Determine the set inclusions among the following languages. In each case, describe the strings belonging to the language.

   (i) $a + b^*$.

   (ii) $a^* + b^*$.

   (iii) $(a^* + b^*)^*$.

3. Describe the following languages in words:

   (i) a^*b^*.

   (ii) $(ab)^*$.

   (iii) $(a + b)(aa + ab + ba + bb)^*$.

   (iv) $(a^2 + b^2)(a + b)^*$.

   (v) $(a + b)^*(a^2 + b^2)(a + b)^*$.

   (vi) $(a + b)^*(a^2 + b^2)$.

   (vii) $(a + b)^*a^2(a + b)^*b^2(a + b)^*$.

4. Let L be any language. Show that if $x, y \in L^*$ then $xy \in L^*$.

   This is an important property of the Kleene star operation.

5. Let $L \subseteq A^*$. Verify the following:

   (i) $(L^*)^* = L^*$.

   (ii) $L^*L^* = L^*$.

   (iii) $L^*L + \varepsilon = L^* = LL^* + \varepsilon$.

   Is it always true that $LL^* = L^*$?

6. Prove that the following hold for all languages L, M, and N.

   (i) $L(MN) = (LM)N$.

   (ii) $L(M + N) = LM + LN$ and $(M + N)L = ML + NL$.

   (iii) If $L \subseteq M$ then $NL \subseteq NM$ and $LN \subseteq MN$.

   (iv) Prove a more general version of (ii) above: products distribute over *arbitrary* (not just finite) unions.

7. Let L, M, N be languages over A. Show that $L(M \cap N) \subseteq LM \cap LN$. Using $A = \{a, b\}$, show that the reverse inclusion does not hold in general by finding a counterexample.

8. Let $A = \{a, b\}$. Show that

$$(ab)^+ = (aA^* \cap A^*b) \setminus (A^*aaA^* + A^*bbA^*).$$

9. Let A be an alphabet and let $u, v \in A^*$. Prove that $uA^* \cap vA^* \neq \emptyset$ if and only if u is a prefix of v or vice versa; when this happens explicitly calculate $uA^* \cap vA^*$.

10. For which languages L is it true that $L^* = L^+$?

11. Let S be a monoid with identity 1. A *submonoid* T of S is a subset such that $1 \in T$ and if $a, b \in T$ then $ab \in T$. Let $L \subseteq A^*$. Prove that L^* is the smallest submonoid of A^* containing L.

    By 'smallest' I mean that if T is any submonoid of A^* containing L then $L^* \subseteq T$.

12. Is $\mathsf{P}(A^*)$ a monoid?

    A *zero* in a semigroup S is an element z such that $zs = z = sz$ for all $s \in S$. Show that if a semigroup has a zero then it has a unique zero.

    Does $\mathsf{P}(A^*)$ contain a zero?

1.4 Finite automata: motivation

An information-processing machine transforms inputs into outputs. In general, there are two alphabets associated with such a machine: an *input alphabet* A for communicating with the machine, and an *output alphabet* B for receiving answers. For example, consider a machine that takes as input sentences in English and outputs the corresponding sentence in Russian.

There is however another way of processing strings, which will form the subject of this course. As before, there is an input alphabet A but this time each input string causes the machine to output either 'yes' or 'no.' Those input strings from A^* that cause the machine to output 'yes' are said to be *accepted* by the machine, and those strings that cause it to output 'no' are said to be *rejected*. In this way, A^* is partitioned into two subsets: the 'yes' subset we call the *language accepted by the machine*, and the 'no' subset we call the *language rejected by the machine.* A machine that operates in this way is called an *acceptor.*

Our aim is to build a mathematical model of a special class of acceptors. Before we give the formal definition in Section 1.5 we shall motivate it by thinking about real machines and then abstracting certain of their features to form the basis of our model.

To be concrete, let us think of two extremes of technology for building an acceptor and find out what they have in common. In Babbage's day the acceptor would have been constructed out of gear-wheels rather like Babbage's 'analytical engine,' the Victorian prototype of the modern computer; in our day, the acceptor would be built from electronic components. Despite their technological differences, the two different types of component involved, gear-wheels in the former and electronic components in the latter, have something in common: they can only do a limited number of things. A gear-wheel can only be in a finite number of positions, whereas many basic electronic components can only be either 'on' or 'off.' We call a specific configuration of gear-wheels or a specific configuration of on-and-off devices a *state*. For example, a clock with only an hour-hand and a minute-hand has 12×60 states that are made visible by the position of the hands. What all real devices have in common is that the total number of states is *finite*. How states are represented is essentially an engineering question.

After a machine has performed a calculation the gear-wheels or electronic components will be in some state dependent on what was being calculated. We therefore need a way of resetting the machine to an initial state; think of this as wiping the slate clean to begin a new calculation.

Every machine should do its job reliably and automatically, and so what the machine does next must be completely determined by its current state and current input, and because the state of a machine contains all the information about the configurations of all the machine's components, what a machine does next is to change state.

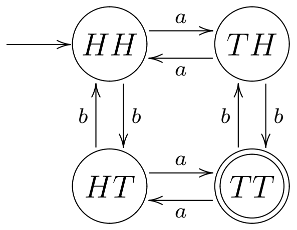
We can now explain how our machine will process an input string $a_1 \ldots a_n$. The machine is first re-initialised so that it is in its initial state, which we call s_0. The first letter a_1 of the string is input and this, together with the fact that the machine is in state s_0, completely determines the next state, say s_1. Next the second letter a_2 of the string is input and this, together with the fact that the machine is in state s_1, completely determines the next state, say s_2. This process continues until the last letter of the input string has been read. At this point, the machine is now ready to pass judgement on the input string. If the machine is in one of a designated set of special states called *terminal* states it deems the string to have been accepted; if not, the string is *rejected*.

To make this more concrete, here is a specific example.

Example 1.4.1 Suppose we have two coins. There are four possible ways of placing them in front of us depending on which is heads (H) and which is tails (T):

$$\mathrm{HH}, \mathrm{TH}, \mathrm{HT}, \mathrm{TT}.$$

Now consider the following two operations: 'flip the first coin,' which I shall denote by a and 'flip the second coin,' which I shall denote by b. Assume that initially the coins are laid out as HH. I am interested in all the possible ways of applying the operations a and b so that the coins are laid out as TT. The states of this system are the four ways of arranging the two coins; the initial state is HH and the terminal state is TT. The following diagram illustrates the relationships between the states and the two operations.



I have marked the start state with an inward-pointing arrow, and the terminal state by a double circle. If we start in the state HH and input the string *aba* Then we pass through the following states:

$$\text{HH} \xrightarrow{a} \text{TH} \xrightarrow{b} \text{TT} \xrightarrow{a} \text{HT}.$$

Thus the overall effect of starting at HH and inputting the string *aba* is to end up in the state HT. It should be clear that those sequences of a's and b's are accepted precisely when the number of a's is odd and the number of b's is odd. We can write this language more mathematically as follows:

$$\{x \in (a+b)^*\colon \mid x\mid_a \text{ and } \mid x\mid_b \text{ are odd}\}.$$

$\square$

To summarise, our mathematical model of an acceptor will have the following features:

- A finite set representing the finite number of states of our acceptor.
- A distinguished state called the *initial state* that will be the starting state for all fresh calculations.
- Our model will have the property that the current state and the current input uniquely determine the next state.
- A distinguished set of *terminal* states.

Exercises 1.4

1. This question is similar to Example 1.4.1. Let $A = \{0, 1\}$ be the input alphabet. Consider the set A^3 of all binary strings of length 3. These will be the states. Let 000 be the initial state and 110 the terminal state. Let $a_1a_2a_3$ be the current state and let a be the input symbol. Then the next state is a_2a_3a; so we shift everything along one place to the left, the left-hand bit drops off and is lost and the right-hand bit is the input symbol. Draw a diagram, similar to the one in Example 1.4.1, showing how states are connected by inputs.

1.5 Finite automata and their languages

In Section 1.4, we laid the foundations for the following definition. A *complete deterministic finite state automaton* **A** or, more concisely, a *finite automaton* and sometimes, just for variety, a *machine* is specified by five pieces of information:

$$\mathbf{A} = (S, A, i, \delta, T),$$

where S is a finite set called the *set of states*, A is the finite *input alphabet*, i is a fixed element of S called the *initial state*, δ is a function $\delta \colon S \times A \to S$ called the *transition function*, and T is a subset of S called the set of *terminal states* (also called *final state*). The phrase 'finite state' is self-explanatory. The meanings of 'complete' and 'deterministic' will be explained below.

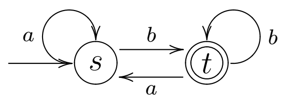
Remark The definition makes sense without the restriction to a finite number of states although this course will deal almost exclusively with the finite state case.

There are two ways of providing the five pieces of information needed to specify an automaton: 'transition diagrams' and 'transition tables.'

A *transition diagram* is a special kind of directed labelled graph: the vertices are labelled by the states S of **A**; there is an arrow labelled a from the vertex labelled s to the vertex labelled t precisely when $\delta(s, a) = t$ in **A**. That is to say, the input a causes the automaton **A** to change from state s to state t. Finally, the initial state and terminal states are distinguished in some way: we mark the initial state by an inward-pointing arrow, $\longrightarrow \bigcirc\!\!\!\!\!\;i$, and the terminal states by double circles $\circledcirc\!\!\!\!\!\;t$.[5]

[5]Another convention is to use outward-pointing arrows to denote terminal states, and

Example 1.5.1 Here is a simple example of a transition diagram of a finite automaton.



We can easily read off the five ingredients that specify an automaton from this diagram:

- The set of states is $S = \{s, t\}$.
- The input alphabet is $A = \{a, b\}$.
- The initial state is s.
- The set of terminal states is $\{t\}$.

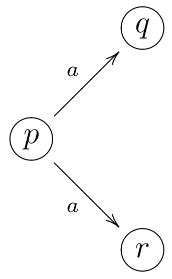
Finally, the transition function $\delta \colon S \times A \to S$ is given by

$$\delta(s, a) = s, \quad \delta(s, b) = t, \quad \delta(t, a) = s, \text{ and } \delta(t, b) = t.$$

$\square$

In order to avoid having too many arrows cluttering up a diagram, the following convention will be used: if the letters $a_1, \ldots, a_m$ label m transitions from the state s to the state t then we simply draw *one* arrow from s to t labelled $a_1, \ldots, a_m$ rather than m arrows labelled a_1 to a_m, respectively.

For a diagram to be the transition diagram of an automaton, two important points need to be borne in mind, both of which are consequences of the fact that $\delta \colon S \times A \to S$ is a function. First, it is impossible for two arrows to leave the same state carrying the same label. Thus a configuration such as



double-headed arrows for states that are both initial and terminal.

is forbidden. This is what we mean by saying that our machines are *deterministic*: the action of the machine is completely determined by its current state and current input and no choice is allowed. Second, in addition to being deterministic, there must be an arrow leaving a given state for each of the input letters; there can be no missing arrows. For this reason we say that our machines are *complete*. Incomplete automata will be defined in Section 2.2, and non-deterministic automata, which need be neither deterministic nor complete, will be defined in Section 3.2.

A *transition table* is just a way of describing the transition function δ in tabular form and making clear in some way the initial state and the set of terminal states. The table has rows labelled by the states and columns labelled by the input letters. At the intersection of row s and column a we put the element $\delta(s, a)$. The states labelling the rows are marked to indicate the initial state and the terminal states. Here is the transition table of our automaton in Example 1.5.1:

	a	b
$\rightarrow s$	s	t
$\leftarrow t$	s	t

We shall designate the initial state by an inward-pointing arrow $\rightarrow$ and the terminal states by outward-pointing arrows $\leftarrow$. If a state is both initial and terminal, then the inward and outward pointing arrows will be written as a single double-headed arrow $\leftrightarrow$.

Notation There is a piece of notation we shall frequently use. Rather than write $\delta(s, a)$ we shall write $s \cdot a$.

When you design an automaton, it really must be an automaton. This means that you have to check that the following two conditions hold:

- There is exactly one initial state.
- For each state s and each input letter $a \in A$, there is *exactly one* arrow starting at s finishing at $s \cdot a$ and labelled by a.

An automaton that satisfies these two conditions — and has a finite number of states, which is rarely an issue when designing an automaton — is said to be *well-formed*.

One thing missing from our definition of a finite automaton is how to process input strings rather than just input letters. Let $\mathbf{A} = (S, A, i, \delta, T)$ be a finite automaton, let $s \in S$ be an arbitrary state, and let $x = a_1 \ldots a_n$ be an arbitrary string. If $\mathbf{A}$ is in state s and the string x is processed then the behaviour of $\mathbf{A}$ is completely determined: for each symbol $a \in A$ and each state s' there is exactly one transition starting at s' and labelled a. Thus we pass through the states $s \cdot a_1$, $(s \cdot a_1) \cdot a_2$ and so on finishing at the state $t = (\ldots((s \cdot a_1) \cdot a_2) \ldots) \cdot a_m$. Thus there is a unique path in $\mathbf{A}$ starting at s and finishing at t and labelled by the symbols of the string $a_1 \ldots a_n$ in turn.

We can formalise this idea by introducing a new function δ^*, called the *extended transition function*. The function δ^* is the unique function from $S \times A^*$ to S satisfying the following three conditions where $a \in A$, $w \in A^*$ and $s \in S$:

(ETF1) $\delta^*(s, \varepsilon) = s$.

(ETF2) $\delta^*(s, a) = \delta(s, a)$.

(ETF3) $\delta^*(s, aw) = \delta^*(\delta(s, a), w)$.

I have claimed that there is a unique function satisfying these three conditions. This will probably seem obvious but does need proving. A proof can be found in my book.

Notation I shall usually write $s \cdot w$ instead of $\delta^*(s, w)$ to simplify notation.

Remark By induction it can be proved that

$$s \cdot (xy) = (s \cdot x) \cdot y$$

for all states s and strings x and y.

It is important to take note of condition (ETF1): this says that the empty string has no effect on states. Thus for each state s we have that $s \cdot \varepsilon = s$.

We can now connect languages and finite automata together. Let

$$\mathbf{A} = (S, A, i, \delta, T)$$

be a complete deterministic automaton. Define the *language accepted* or *recognised* by $\mathbf{A}$, denoted $L(\mathbf{A})$, to be

$$L(\mathbf{A}) = \{w \in A^*\colon i \cdot w \in T\}.$$

A language is said to be *recognisable* if it is recognised by some finite automaton.

The language recognised by an automaton $\mathbf{A}$ with input alphabet A therefore consists of all strings in A^* that label paths in $\mathbf{A}$ starting at the initial state and concluding at a terminal state. There is only one string where we have to think a little to decide whether it is accepted or not. This is the empty string. Suppose first that $\varepsilon \in L(\mathbf{A})$. If i is the initial state of $\mathbf{A}$ then by definition $i \cdot \varepsilon$ is terminal because $\varepsilon \in L(\mathbf{A})$, and so i is terminal. Now suppose that the initial state i is also terminal. Because $i = i \cdot \varepsilon$, it follows from the definition that $\varepsilon \in L(\mathbf{A})$. *We see that the empty string is accepted by an automaton if and only if the initial state is also terminal.* This is a small point but worth remembering.

Example 1.5.2 We describe the language recognised by our machine in Example 1.5.1. We have to find all those strings in $(a + b)^*$ that label paths starting at s and finishing at t. First, any string x ending in a 'b' will be accepted. To see why let $x = x'b$ where $x' \in A^*$. If x' leads the machine to state s, then the b will lead the machine to state t; and if x' leads the machine to state t, then the b will keep it there. Second, a string x ending in 'a' will not be accepted. To see why let $x = x'a$ where $x' \in A^*$. If x' leads the machine to state s, then the a will keep it there; and if x' leads the machine to state t, then the a will send it to state s. Finally, the empty string is not accepted by this machine because the initial state is not terminal. We conclude that $L(\mathbf{A}) = A^*b$. □

Remark Let X be a set and S a monoid. By a *right action* of S on X we mean a function $X \times S \to X$ mapping (x, s) to $x \cdot s$ satisfying two conditions: first, $x \cdot 1 = x$ for all $x \in X$ and second, $x \cdot (st) = (x \cdot s) \cdot t$ for all $s, t \in S$ and $x \in X$. Thus underlying every finite automaton is an action of a free monoid on a finite set.

Exercises 1.5

1. For each of the following transition tables construct the corresponding transition diagram.

(i)

	a	b
$\rightarrow s_0$	s_1	s_0
s_1	s_2	s_1
$\leftarrow s_2$	s_0	s_2

(ii)

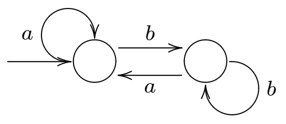
	a	b
$\leftrightarrow s_0$	s_1	s_1
s_1	s_0	s_2
$\leftarrow s_2$	s_0	s_1

(iii)

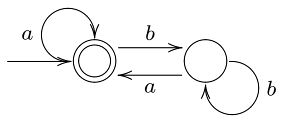
	a	b	c
$\leftrightarrow s_0$	s_1	s_0	s_2
s_1	s_0	s_3	s_0
$\leftarrow s_2$	s_3	s_2	s_0
$\leftarrow s_3$	s_1	s_0	s_1

2. Determine which of the following diagrams are finite automata and which are not, and give reasons for your answers. The alphabet in question is $A = \{a, b\}$.

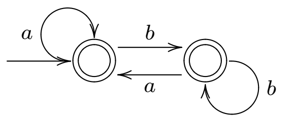
(i)



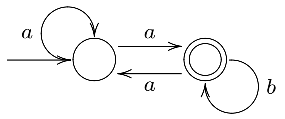
(ii)



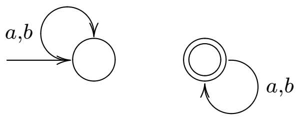
(iii)



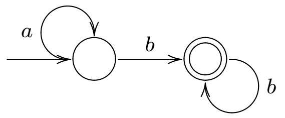
(iv)



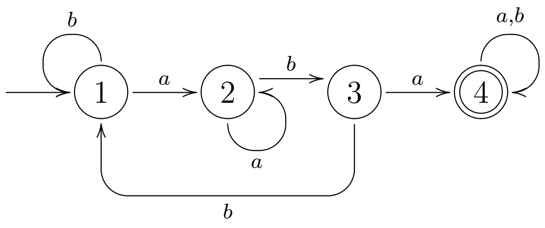
(v)



(vi)



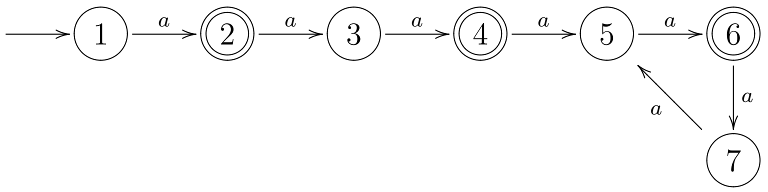
3. Let $A = \{a, b\}$ with the order $a < b$. Consider the automaton below:



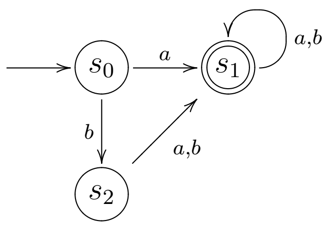
Draw up the following table: the rows should be labelled by the states, and the columns by all strings x in A^* where $0 \leq |x| \leq 3$ written in tree order. If q is a row and x is a column then the entry in the q-th row and x-th column should be the state $q \cdot x$.

4. For each of the automata below, describe the language recognised.

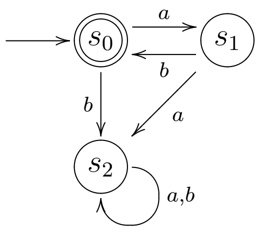
(i)



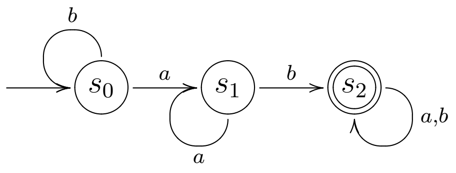
(ii)



(iii)



(iv)



5. This question is intended to get you thinking. Find an automaton that represents all those strings over the alphabet $\{0, 1\}$ that represents a number in binary that is a multiple of three. The empty string represents the number 0. The numbers 11, 011, 0011 and so on are all to be regarded as the same; thus leading zeros are permitted.

6. (i) What is meant by a *decision problem*?

   (ii) Explain how decision problems give rise to languages.

   (iii) Under what circumstances is a decision problem (or language) *decidable*?

   (iv) Using the usual encoding of simple graphs, describe the language associated with the decision problem 'is the graph complete?' (A simple graph is complete if every vertex is joined to every other vertex). Is this language decidable? Explain.

   (v) Are recognisable languages decidable? Explain.

1.6 Summary of Chapter 1

- *Alphabets*: An alphabet is any finite set. The elements of an alphabet are called symbols or letters.

- *Strings*: A string is any finite sequence of symbols taken from a fixed alphabet. The empty string is denoted ε. The set of all strings taken from the alphabet A is A^*, and the set of all non-empty strings is A^+.

- *Languages*: A language over an alphabet A is any subset of A^*; this includes the two extremal subsets: the empty set $\emptyset$ and A^* itself.

- *Language operations*: There are a number of important ways of combining languages L and M to form new languages. The Boolean operations $L \cap M$, $L + M$ and L' are, respectively, intersection, union, and complementation. There are two further operations $L \cdot M$ and L^*, which are, respectively, product and Kleene star. The product $L \cdot M$ of two languages, usually written just LM, consists of all strings that can be written as a string in L followed by a string in M. The Kleene star L^* of a language consists of the empty string and all strings that can factorised as products of strings in L. The set $\mathsf{P}(A^*)$ of all languages over the alphabet A forms what is known as an *idempotent semiring*: with respect to $+$ the set of languages forms a commutative monoid, with identity $\emptyset$, in which every element is idempotent, meaning that $A + A = A$, and with respect to $\cdot$ the set of languages forms a monoid with identity $\{\varepsilon\}$. In addition $\cdot$ distributes over $+$.

- *Finite automata*: These are special kinds of algorithms for deciding the membership of languages. They consist of a finite number of states, an input alphabet A, a distinguished initial state, a finite number of transitions labelled by the elements of A, and a finite set of terminal states. In addition, they are complete and deterministic. The language recognised or accepted by an automaton consists of all strings over A that label paths from the initial state to one of the terminal states. Completeness and determinism imply that each input string labels a unique path starting at the initial state.

Chapter 2

Recognisable languages

In Chapter 1, we introduced finite automata and the languages they recognise. In this chapter, we look at ways of constructing certain kinds of automata. We also prove that there are languages that are *not* recognisable.

2.1 Designing automata

Designing an automaton **A** to recognise a language L is more an art than a science. However, it is possible to lay down some guidelines. In this section, I will describe the general points that need to be born in mind, and in Sections 2.2 to 2.6, I will describe some specific techniques for particular kinds of languages.

Automata can be regarded as simple programming languages, and so the methods used to write programs can be adopted to help design automata. Whenever you write a program to solve a problem, it is good practice to begin by formulating the algorithm that the program should implement. By the same token, before trying to construct an automaton to recognise a language, you should first formulate an algorithm that accomplishes the task of recognising the language. There is however an important constraint: your algorithm must only involve using a fixed amount of memory. One way of ensuring this is to imagine how you would set about recognising the strings belonging to a language for *extremely large* inputs. When you have done this, you can then implement your algorithm by means of an automaton.

Once you have a design, **A**, it is easy to check that it is well-formed — this is equivalent to checking that a program is syntactically correct — but the crucial point now is to verify that your automaton really recognises the language L in question. This involves checking that two conditions hold:

(1) Each string accepted by $\mathbf{A}$ is in L.

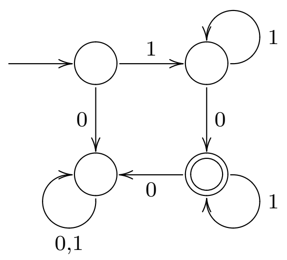
(2) Each string in L is accepted by $\mathbf{A}$.

I have emphasised that two conditions must hold, because it is a very common mistake to check only (1). If both of these conditions are satisfied then you have solved the problem. But if either one of them is violated then you have to go back and modify your machine and try again. Unfortunately, it is easier to show that your automaton *does not* work than it is to show it *does*. To show that your machine is wrong it is enough to find just one string x that is in the language L but not accepted by the machine $\mathbf{A}$, or is accepted by the machine $\mathbf{A}$ but is not in the language L. The difficulty is that A^* contains infinitely many strings and your machine has to deliver the correct response to each of them.

The minimum you should do to show that your well-formed automaton solves the problem is to try out some test strings on it, making sure to include both strings belonging to the language and those that do not. However even if your automaton passes these tests with flying colours it could still be wrong. There are two further strategies that you can use. First, if you find you have made a mistake then try to use the nature of the mistake to help you see how to correct the design. Second, try to be clear about the function of each state in your machine: each state should be charged with detecting some feature of that part of the input string that has so far been read. Some of the ideas that go into constructing automata are illustrated by means of examples in the following sections. At the same time I shall describe some useful techniques for constructing automata. Although I often use alphabets with just two letters, this is just for convenience, similar examples can be constructed over other alphabets.

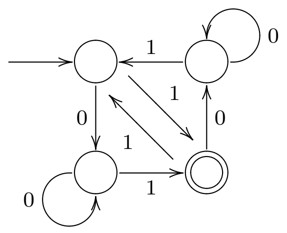
Exercises 2.1

1. Let **A** be the automaton:



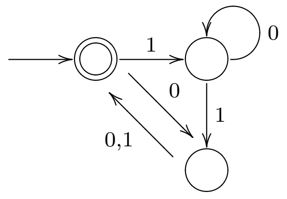
and let $L = 1^+0$. Show that $L \subseteq L(\mathbf{A})$, but that the reverse inclusion does not hold. Describe $L(\mathbf{A})$.

2. Let **A** be the automaton:



Show that every string in $L(\mathbf{A})$ has an odd number of 1's, but that not every string with an odd number of 1's is accepted by **A**. Describe $L(\mathbf{A})$.

3. Let **A** be the automaton:



Let L be the language consisting of all strings over $\{0, 1\}$ containing an odd number of 1's. Show that neither $L \subseteq L(\mathbf{A})$ nor $L(\mathbf{A}) \subseteq L$. Describe $L(\mathbf{A})$.

2.2 Automata over one letter alphabets

In this section, we shall describe the recognisable languages over a one letter alphabet $A = \{a\}$.

Theorem 2.2.1 *If a language $L \subseteq a^*$ is recognisable then*

$$L = X + Y(a^p)^*,$$

where X and Y are finite sets and $p \geq 0$.

Proof Let L be recognisable. Because the alphabet contains only one letter, an automaton recognising L must have a particular form, which we now describe. Let the initial state be q_1. Then either $q_1 \cdot a = q_1$ in which case q_1 is the only state, or $q_1 \cdot a$ is some other state, q_2 say. For each state q, either $q \cdot a$ is a previously constructed state or a new state. Since the automaton is finite there must come a point where $q \cdot a$ is a previously occurring state. It follows that an automaton recognising L consists of a *stem* of s states $q_1, \ldots, q_s$, and a *cycle* of p states $r_1, \ldots r_p$ connected together as follows:

$$\begin{tikzcd}
{} \arrow[r] & q_1 \arrow[r, "a"] & q_2 \arrow[r, dashed] & q_s \arrow[r, "a"] & r_1 \arrow[r, "a"] & r_2 \arrow[r, dashed] & r_p \arrow[ll, bend right, "a"']
\end{tikzcd}$$

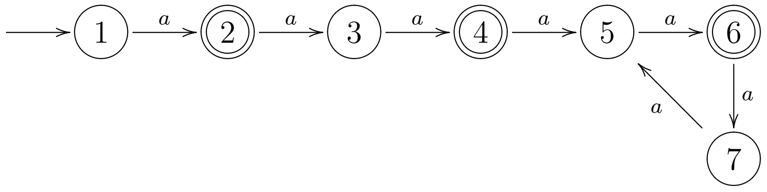
The terminal states therefore form two sets: the terminal states T' that occur in the stem and the terminal states T'' that occur in the cycle. Let X be the set of strings recognised by the stem states: each string in X corresponds to exactly one terminal state T' in the stem. Let T'' consist of n terminal states, which we number 1 to n. For each terminal state i let y_i be the shortest string required to reach it from q_1. Then $y_i(a^p)^*$ is recognised by the automaton for all $1 \leq i \leq n$. Put $Y = \{y_i \colon 1 \leq i \leq n\}$. Then the language recognised by the automaton is $X + Y(a^p)^*$. □

Remark The sets X and Y satisfy some extra conditions. What are they?

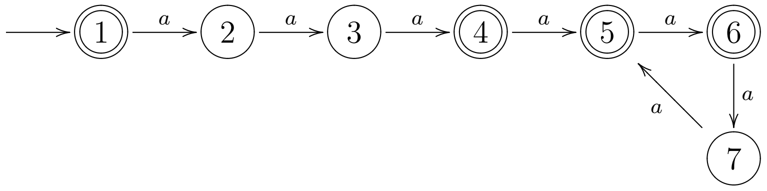
Exercises 2.2

1. Describe the languages recognised by the following automata.

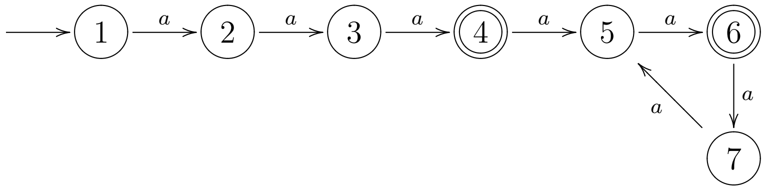
(i)



(ii)



(iii)



2. Construct automata recognising the following languages.

   (i) $(a^2 + a^5) + (a^2 + a^3)(a^4)^*$.

   (ii) $(a^2 + a^4) + (a^2 + a^4)(a^2)^*$.

   (iii) $(a^2 + a^5) + (a^2 + a^4)(a^3)^*$.

3. Is the converse to Theorem 2.2.1 true? That is, if $L \subseteq a^*$ is such that

$$L = X + Y(a^p)^*,$$

   where X and Y are finite sets and $p \geq 0$ is it true that L is recognisable?

4. Show that $(a^2 + a^3)(a^3)^* + (a + a^2)(a^4)^*$ is recognisable.

5. What is the longest string not in the language $(a^3 + a^5)^*$?

6. What is the longest string not in the language $(a^p + a^q)^*$ where p and q are coprime (meaning that the largest integer dividing them both is 1).

7. Let $\{n_1, n_2, n_3, \ldots\}$ be a set of natural numbers. We say that this subset is *1-recognisable* if the language $\{a^{n_i} \colon i = 1, 2, 3, \ldots\}$ is recognisable over the one letter alphabet $A = \{a\}$. (The meaning of the '1' is 'base 1'.) A subset S of $\mathbb{N}$ is said to be *ultimately periodic* if there exists $n \geq 0$ and a $p > 0$ such that for all $m \geq n$ we have that $m \in S$ iff $m + p \in S$. Prove that $S \subseteq \mathbb{N}$ is 1-recognisable iff it is ultimately periodic.

2.3 Incomplete automata

A useful design technique is illustrated by the following example.

Example 2.3.1 Construct an automaton to recognise the language $L = \{abab\}$. We first construct a 'spine' as follows:

$$\begin{tikzcd}
{} \arrow[r] & 1 \arrow[r, "a"] & 2 \arrow[r, "b"] & 3 \arrow[r, "a"] & 4 \arrow[r, "b"] & 5
\end{tikzcd}$$

This diagram has a path labelled *abab* from the initial state to the terminal state and so we have ensured that the string *abab* will be recognised. However, it fails to be an automaton because there are missing transitions. It is tempting to put loops on the states and label the loops with the missing symbols, but this is exactly the wrong thing to do (why?). Instead, we add a new state, called a 'sink state,' to which we can harmlessly lead all unwanted transitions. In this way, we obtain the following automaton.

$$\begin{tikzcd}
{} \arrow[r] & 1 \arrow[r, "a"] \arrow[drr, "b"] & 2 \arrow[r, "b"] \arrow[dr, "a"] & 3 \arrow[r, "a"] \arrow[d, "b"] & 4 \arrow[r, "b"] \arrow[dl, "a"] & 5 \arrow[dll, "{a,b}"] \\
& & & 6 \arrow[loop below, "{a,b}"] & &
\end{tikzcd}$$

$\square$

The idea behind the above example can be generalised. The 'spine' we constructed above is an example of an incomplete automaton; this is just like an automaton except that there are missing transitions. More formally, we define them as follows. An *incomplete automaton* is defined in exactly the same way as a complete deterministic automaton except that the transition *function* δ is replaced by a *partial function*. This means that $\delta(s,a)$ is not defined for some $(s,a) \in S \times A$; in such cases, we say that the machine *fails*.

Let $\mathbf{A} = (S, A, i, \delta, T)$ be an incomplete automaton. To define the language accepted by this machine we proceed as in the complete case, and with a similar justification. The extended transition function δ^* is defined as before, except this time δ^* is a partial function from $S \times A^*$ to S. More precisely, δ^* is defined as follows. Let $a \in A$, $w \in A^*$ and $s \in S$:

(ETF1) $\delta^*(s, \varepsilon) = s$.

(ETF2) $\delta^*(s, a) = \delta(s, a)$.

(ETF3) $\delta^*(s, aw) = \begin{cases} \delta^*(\delta(s,a), w) & \text{if } \delta(s,a) \text{ is defined} \\ \text{not defined} & \text{else.} \end{cases}$

We now define $L(\mathbf{A})$ to consist of all those strings $x \in A^*$ such that $\delta^*(i, x)$ is defined and is a terminal state.

In a complete automaton, there is *exactly one* path in the machine starting at the initial state and labelled by a given string. In an incomplete automaton, on the other hand, there is *at most one* path starting at the initial state and labelled by a given string. The language recognised by an incomplete automaton still consists of all strings over the input alphabet that label paths beginning at the initial state and ending at a terminal state.

It is easy to convert an incomplete machine into a complete machine that recognises the same language.

Proposition 2.3.2 *For each incomplete automaton* $\mathbf{A}$ *there is a complete automaton* $\mathbf{A}^c$ *such that* $L(\mathbf{A}^c) = L(\mathbf{A})$.

Proof Let $\mathbf{A} = (S, A, i, \delta, T)$. Define $\mathbf{A}^c$ as follows. Let ∞ be any symbol not in S. Then $S \cup \{\infty\}$ will be the set of states of $\mathbf{A}^c$; its initial state is i and its set of terminal states is T. The transition function of γ of $\mathbf{A}^c$ is defined as follows. For each $a \in A$ and $s \in S \cup \{\infty\}$

$$\gamma(s,a) = \begin{cases} \delta(s,a) & \text{if } \delta(s,a) \text{ is defined} \\ \infty & \text{if } s \neq \infty \text{ and } \delta(s,a) \text{ is not defined} \\ \infty & \text{else.} \end{cases}$$

Because the machine $\mathbf{A}$ is sitting inside $\mathbf{A}^c$, it is immediate that $L(\mathbf{A}) \subseteq L(\mathbf{A}^c)$. To prove the reverse inclusion, observe that any string that is accepted by $\mathbf{A}^c$ cannot pass through the state ∞ at any point. Thus the string is essentially being processed by $\mathbf{A}$. $\square$

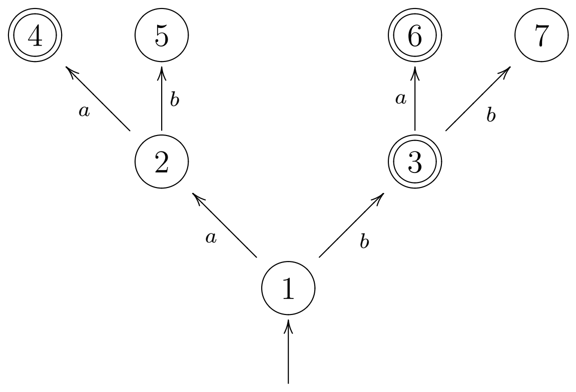
We say that a state s in an automaton is a *sink state* if $s \cdot a = s$ for each $a \in A$ in the input alphabet. Thus the state ∞ in our construction above is a sink state, and the process of converting an incomplete machine into a complete machine is called *completion (by adjoining a sink state).* The automaton $\mathbf{A}^c$ is called the *completion* of $\mathbf{A}$.

It is sometimes easier to design an incomplete automaton to recognise a language and than to complete it by adjoining a sink state then to try to design the automaton all in one go. We can apply this idea to show that any *finite* language is recognisable. We illustrate this result by means of an example.

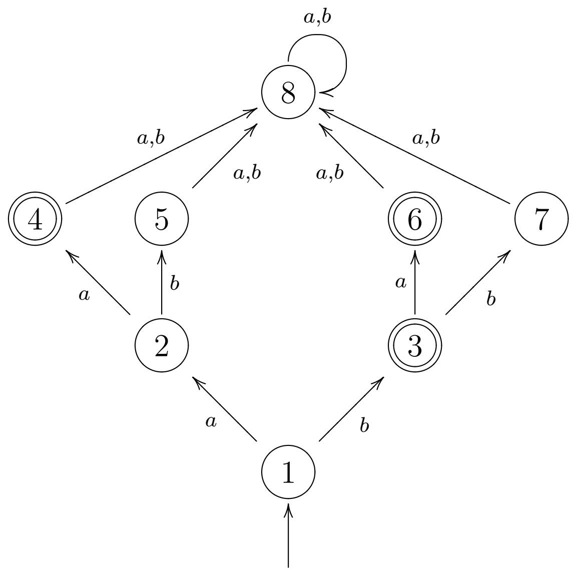
Example 2.3.3 Consider the finite language $\{b, aa, ba\}$. The starting point for our automaton is the part of the tree over $\{a, b\}$, which contains all strings of length 2 and smaller:

$$\begin{tikzcd}
aa & ab & & ba & bb \\
& a \arrow[ul, "a"'] \arrow[u, "b"'] & & b \arrow[u, "a"] \arrow[ur, "b"'] & \\
& & \varepsilon \arrow[ul, "a"] \arrow[ur, "b"'] & &
\end{tikzcd}$$

Notice that I have used labelled arrows rather than edges. This is used to build an incomplete automaton that recognises $\{b, aa, ba\}$: the vertices of the tree become the states, the initial state is the vertex labelled ε, and the terminal states are the vertices labelled with the strings in the language.

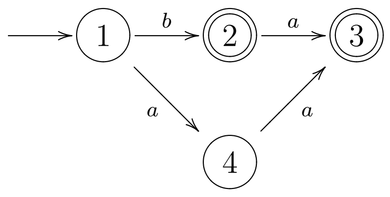


This incomplete automaton is then completed by the addition of a sink state. We thus obtain the following automaton that recognises $\{b, aa, ba\}$.



$\square$

Our example of an automaton that recognises the language $\{b, aa, ba\}$ raises another point. Another (incomplete) machine that recognises this language is



Thus by adjoining a sink state, we need only 5 states to recognise $\{b, aa, ba\}$ instead of the 8 in our example above. The question of finding the smallest number of states to recognise a given language is one that we shall pursue in Chapter 6.

The proof of the following is now left as an exercise.

Proposition 2.3.4 *Every finite language is recognisable.* □

Exercises 2.3

1. Construct an automaton to recognise the language

$$L = \{\varepsilon, ab, a^2b^2, a^3b^3\}.$$

2. Write out a full proof of Proposition 2.3.4.

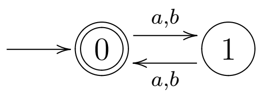
2.4 Automata that count

Counting is one of the simplest ways of describing languages. For example, we might want to describe a language by restricting the lengths of the strings that can appear, or by requiring that a particular letter or pattern appears a certain number of times. We shall also see that there are limits to what automata can do in the realm of counting. We begin with a simple example.

Example 2.4.1 Construct an automaton to recognise the language

$$L = \{x \in (a+b)^* \colon |\, x\,| \text{ is even }\}.$$

The first step in constructing an automaton is to ensure that you understand what the language is. In this case, $x \in L$ precisely if $|\, x\,| = 0, 2, 4, \ldots$. The empty string ε is accepted since $|\, \varepsilon\,| = 0$, and so the initial state will also have to be terminal. In this case, we shall only need two states: one state remembers that we have read an even number of symbols and another that remembers that we have read an odd number of symbols. We therefore obtain the following automaton.



If, instead, we wanted to construct an automaton that recognised the language of strings over $\{a, b\}$ of *odd* length, then we would simply modify the above machine by making state 0 non-terminal and state 1 terminal. $\square$

To generalise the above example, I shall need some terminology. The set of integers, that is the set of positive and negative whole numbers, is denoted $\mathbb{Z}$. The word 'number' will almost always mean 'integer' from now on. If $a, b \in \mathbb{Z}$ we say that a *divides* b, or that b is *divisible by* a, or that b is a *multiple* of a, if $b = aq$ for some $q \in \mathbb{Z}$; this is written mathematically as $a \mid b$. If $a, b \in \mathbb{Z}$ and $a > 0$ then we can write $b = aq + r$ where $0 \leq r < a$. The number q is called the *quotient* and r is called the *remainder*. The quotient and remainder are uniquely determined by a and b meaning that if $b = aq' + r'$ where $0 \leq r' < a$ then $q = q'$ and $r = r'$. This result is called the 'Remainder Theorem' and is one of the basic properties of the integers.

Using this terminology, let us look again at odd and even numbers. If we divide a number by 2, then there are exactly two possible remainders: 0 or 1. A number that has no remainder when divided by 2 is just an even number and a number that leaves the remainder 1 when divided by 2 is just an odd number. It is an accident of history that English, and many other languages, happen to have single words that mean 'leaves no remainder when divided by 2' and 'leaves remainder 1 when divided by 2.'

Now let us look at what happens when we divide a number by 3. This time there are three possible cases: 'leaves no remainder when divided by 3,' 'leaves remainder 1 when divided by 3,' and 'leaves remainder 2 when divided by 3.' In this case, there are no single words in English that we can use to substitute for each of these phrases, but this does not matter.

Let $n \geq 2$ be an integer, and let a and b be arbitrary integers. We say that a is *congruent to b modulo n*, written as

$$a \equiv b \pmod{n},$$

if $n \mid (a-b)$. An equivalent way of phrasing this definition is to say that a and b have the same remainder when divided by n. Put $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$, the set of possible remainders when a number is divided by n.

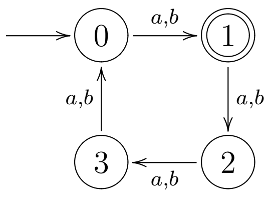
Using this notation, we see that a is even precisely when $a \equiv 0\ (\mathrm{mod}2)$ and is odd when $a \equiv 1 \pmod 2$. If $a \equiv b \pmod 2$ then we say they have the *same parity*: they are either both odd or both even. If a number $a \equiv 0 \pmod n$ then it is divisible by n.

Now that we have this terminology in place, we can generalise Example 2.4.1.

Example 2.4.2 Construct an automaton recognising the language

$$L = \{x \in (a+b)^*\colon |x| \equiv 1 \pmod 4\}.$$

In this case, a string x is in L if its length is $1, 5, 9, 17, \ldots$. In other words, it has length one more than a multiple of 4. Notice that we are not interested in the exact length of the string. It follows that we must reject strings that have lengths $4q$, $4q+2$, or $4q+3$ for some q; we do not need to worry about strings of length $4q+4$ because that is itself a multiple of 4. In other words, there are only four possibilities, and these four possibilities will be represented by four states in our machine. I will label them $0, 1, 2$, and 3, where the state r means 'the length of the string read so far is $4q+r$ for some q.' The automaton that recognises L is therefore as follows:



□

It should now be clear that we can easily construct automata to recognise any language L of the form,

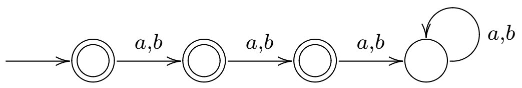
$$L = \{x \in (a+b)^*\colon |\,x\,| \equiv r \pmod n\},$$

for any $n \geq 2$ and $0 \leq r < n$. We now turn to a different kind of counting.

Example 2.4.3 Construct an automaton which recognises the language

$$L = \{x \in (a+b)^*\colon |x| < 3\}.$$

Here we are required to determine length up to some number; this is called 'threshold counting.' We have to accept the empty string, all strings of length 1, and all strings of length 2; we reject all other strings. We are therefore led to the following automaton:



□

In the above example, there is nothing sacrosanct about the number 3. Furthermore, we can easily modify our machine to deal with similar but different conditions on $|x|$ such as $|x| \leq 3$ or $|x| = 3$ or $|x| \geq 3$ or where $|x| > 3$.

Examples 2.4.1, 2.4.2, and 2.4.3 are typical of the way that counting is handled by automata: we can determine length modulo a fixed number, and we can determine length relative to some fixed number.

Our next result shows that there are limits to what we can count using finite automata.

Proposition 2.4.4 *The language*

$$L = \{a^n b^n\colon n \in \mathbb{N}\}$$

is not recognisable.

Proof When we say an automaton $\mathbf{A}$ recognises a language L we mean that it recognises *precisely* the strings in L and no others.

We shall argue by contradiction. Suppose $\mathbf{A} = (S, A, s_0, \delta, T)$ is a finite automaton such that $L = L(\mathbf{A})$. Let

$$q_n = s_0 \cdot a^n \text{ and } t_n = q_n \cdot b^n,$$

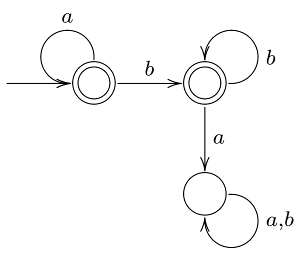
where $n \geq 0$. Thus q_n is the name we give the state that we reach when starting in the initial state s_0 and inputting the string a^n; and t_n is the name we give the state that we reach when starting in q_n and inputting b^n. Then

$s_0 \cdot (a^n b^n) = t_n$ and so t_n is a terminal state because $a^n b^n \in L$. We claim that if $i \neq j$ then $q_i \neq q_j$. Suppose to the contrary that $q_i = q_j$ for some $i \neq j$. Then

$$s_0 \cdot (a^i b^j) = q_i \cdot b^j = q_j \cdot b^j = t_j.$$

But this would imply $a^i b^j \in L$ and we know $i \neq j$. Since this cannot happen, we must have $i \neq j$ implies $q_i \neq q_j$ and so **A** has infinitely many states. This is a contradiction. □

The problem with the language $\{a^n b^n : n \in \mathbb{N}\}$ is that we have to compare the number of a's with the number of b's and there can be an arbitrary number of both. Notice that we can construct an automaton that recognises $a^* b^*$:



Thus an automaton can check that all the a's precede all the b's.

Exercises 2.4

1. Let $A = \{a, b\}$. Construct finite automata for the following languages.

   (i) All strings x in A^* such that $|x| \equiv 0 \pmod 3$.

   (ii) All strings x in A^* such that $|x| \equiv 1 \pmod 3$.

   (iii) All strings x in A^* such that $|x| \equiv 2 \pmod 3$.

   (iv) All strings x in A^* such that $|x| \equiv 1$ or $2 \pmod 3$.

2. Construct a finite automaton to recognise the language

$$L = \{x \in (a+b)^* : |x|_a \equiv 1 \pmod 5\}.$$

3. Let $A = \{0, 1\}$. Construct finite automata to recognise the following languages.

   (i) All strings x in A^* where $|x| < 4$.

(ii) All strings x in A^* where $|x| \leq 4$.

(iii) All strings x in A^* where $|x| = 4$.

(iv) All strings x in A^* where $|x| \geq 4$.

(v) All strings x in A^* where $|x| > 4$.

(vi) All strings x in A^* where $|x| \neq 4$.

(vii) All strings x in A^* where $2 \leq |x| \leq 4$.

4. Construct a finite automaton to recognise the language

$$\{x \in (a+b)^* \colon |x|_a \leq 4\}.$$

5. Show that the language $\{a^i b^j \colon i \equiv j \pmod 2\}$ is recognisable.

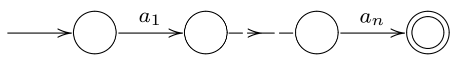
6. Let $A = \{a, b, c\}$. Construct a finite automaton recognising those strings in A^*, where the string *abc* occurs an odd number of times.

2.5 Automata that locate patterns

In this section, we shall show that the languages xA^*, A^*xA^*, and A^*x are all recognisable where A is any alphabet and x is any non-empty string. We begin with the simplest case: we show that the languages xA^* are recognisable.

Proposition 2.5.1 *Let A be an alphabet and let $x \in A^+$ be a string of length n. The language xA^* can be recognised by an automaton with $n+2$ states.*

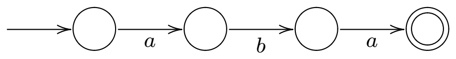
Proof Because $x \in xA^*$ the string x itself must be accepted by any prospective automaton. So if $x = a_1 \ldots a_n$ where each $a_i \in A$, then we must have the following states:



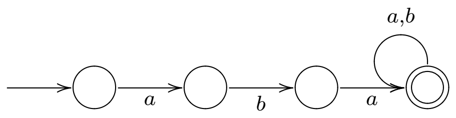
If we now put a loop on the last state labelled with the elements of A, we shall then have an incomplete automaton recognising xA^*. It is now a simple matter to complete this automaton to obtain one recognising the same language. □

We now turn to the problem of showing that languages of the form A^*xA^* are recognisable. This is not quite as straightforward and so we begin with an example to illustrate the ideas involved.

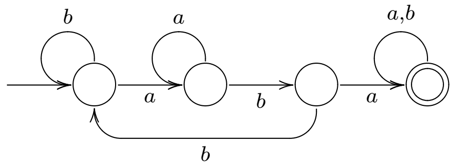
Example 2.5.2 Construct an automaton that recognises the language $L = (a+b)^*aba(a+b)^*$. In other words, all strings that contain aba as a factor. The first point to note is that aba should itself be accepted. So we can immediately write down the spine of the machine:



Once we have ascertained that an input string contains the factor aba we do not care what follows. So we can certainly write



To find out what to do next, put yourself in the position of having to detect the string aba in a very long input string. As a finite automaton you can only read one letter at a time, so imagine that you are constrained to view the input string one letter at a time through a peephole. If you are reading a b, then you are not interested, but as soon as you read an a you are: you make a mental note 'I have just read an a.' If you read a b next, then you get even more interested: you make a mental note 'I have just read ab;' if instead you read an a, then you simply stay in the 'I have just read an a' state. The next step is the crucial one: if you read an a, then you have located the string aba, you do not care what letters you read next; if on the other hand you read a b, then it takes you back to the 'uninterested' state. We see now that the four states on our spine correspond to: 'uninterested,' 'just read an a,' 'just read ab' and 'success!' The automaton we require is therefore the following one:



□

Our example above can be used to formulate a general principle for constructing automata that recognise languages of the form A^*xA^* where $x \in A^+$.

Proposition 2.5.3 *Let A be an alphabet and $x \in A^+$ a string of length n. The language A^*xA^* can be recognised by an automaton with $n+1$ states.*

Proof The first step is to construct the spine as we did in our example. If $x = a_1 \ldots a_n$, then this spine will have $n+1$ states: the first one is initial, the last is terminal, and the transitions are labelled in turn a_i for $i = 1, \ldots, n$; the last state also carries a loop labelled A.

Because we read an input string from left to right, each of these $n+1$ states is really storing which *prefix* of x we have read in the input: from the first state representing ε to the last representing x itself. To work out where to put the missing transitions, suppose that we are in the state corresponding to the prefix y of x, where $y = a_1 \ldots a_i$ and that the next letter of the input string we read is a. There are two cases to consider.

(Case 1): suppose that $a = a_{i+1}$, that is ya is also a prefix of x. Then we simply move to the next state to the right along the spine.

(Case 2): suppose that $a \neq a_{i+1}$. It is tempting to think that we have to go back to the initial state, but this is not necessarily so. The string ya is not a prefix of x; however we can always find a *suffix* of ya that is a *prefix* of x; we do not exclude the possibility that this suffix could be ε. Choose the *longest* suffix z of ya that is a prefix of x. The transition we require is then

$$\begin{tikzcd} \circled{y} \arrow[r, "a"] & \circled{z} \end{tikzcd}$$

More generally, for a fixed string x and arbitrary string u we denote by $\sigma_x(u)$ the longest suffix of u that is a prefix of x. Thus $z = \sigma_x(ya)$.

Notice that (Case 1) is really included in the rule stated in (Case 2) because if ya is a prefix of x then $\sigma_x(ya) = ya$. $\square$

We illustrate the design technique contained in the above result by the following example.

Example 2.5.4 Construct an automaton to recognise the language $A^*ababbA^*$. We construct two tables: the transition table and an auxiliary table that will help us to complete the transition table. We begin by entering in the transition table the transitions on the spine:

	a	b
ε	a	1
a	2	ab
ab	aba	3
aba	4	$abab$
$abab$	5	$ababb$
$ababb$	$ababb$	$ababb$

I have numbered the transitions we still have to find. The auxiliary table below gives the calculations involved in finding them.

	u	proper suffixes of u	$\sigma_{ababb}(u)$
1	b	ε	ε
2	aa	ε, a	a
3	abb	ε, b, bb	ε
4	$abaa$	ε, a, aa, baa	a
5	$ababa$	$\varepsilon, a, ba, aba, baba$	aba

The first column, labelled u, is the concatenation of the prefix labelling the state and the input letter: $\varepsilon \cdot b$, $a \cdot a$, $ab \cdot b$, $aba \cdot a$, and $abab \cdot a$, respectively. The last column, labelled $\sigma_{ababb}(u)$, is the 'longest suffix of the string in the first column, which is a prefix of $ababb$'; we use the middle column to determine this string. We can now complete the transition table using the auxiliary table:

	a	b
$\rightarrow \varepsilon$	a	ε
a	a	ab
ab	aba	ε
aba	a	$abab$
$abab$	aba	$ababb$
$\leftarrow ababb$	$ababb$	$ababb$

It is now easy to draw the transition table of the required automaton. □

Finally, we show that languages of the form A^*x are recognisable.

Proposition 2.5.5 *Let A be an alphabet and $x \in A^+$ a string of length n. The language A^*x can be recognised by an automaton with $n + 1$ states*

Proof The construction is similar to the one contained in Proposition 2.5.3. The first step is to construct the spine. If $x = a_1 \dots a_n$, then this spine will have $n + 1$ states: the first one is initial, the last is terminal, and the transitions are labelled in turn a_i for $i = 1, \dots, n$. In this case, the last state does *not* carry a loop labelled A. We now carry out exactly the same procedure as in Proposition 2.5.3, except this time we apply it also to the terminal state.□

Exercises 2.5

1. Let $A = \{a, b\}$. Construct finite automata to recognise the following languages.

   (i) All strings in A^* that begin with ab.

   (ii) All strings in A^* that contain ab.

   (iii) All strings in A^* that end in ab.

2. Let $A = \{0, 1\}$. Construct automata to recognise the following languages.

   (i) All strings that begin with 01 or 10.

   (ii) All strings that start with 0 and contain exactly one 1.

   (iii) All strings of length at least 2 whose final two symbols are the same.

3. Let $A = \{a, b\}$. Construct an automaton to recognise the language

$$A^*aaA^*bbA^*.$$

4. Let $A = \{a, b, c\}$. Construct an automaton to recognise all strings that begin or end with a double letter.

2.6 Boolean operations

In describing languages we frequently use words such as ‘and’ ‘or’ and ‘not.’ For example, we might describe a language over the alphabet $\{a, b, c\}$ to consist of all strings that satisfy the following condition: they begin with *either* an a *or* a c *and* do *not* contain ccc as a factor. In this section, we describe algorithms for constructing automata where the description of the languages involves Boolean operations.

Example 2.6.1 Consider the language

$$L = \{x \in (a+b)^* \colon |x| \equiv 1 \pmod 4\},$$

of Example 2.4.2. We showed in Section 2.4 how to construct an automaton $\mathbf{A}$ to recognise L. Consider now the language $L' = A^* \setminus L$. We could try to build a machine from scratch that recognises L' but we would much prefer to find some way of adapting the machine $\mathbf{A}$ we already have to do the job. The strings in L' are those $x \in (a+b)^*$ such that $|x| \equiv 0$ or $|x| \equiv 2$ or $|x| \equiv 3$ (mod 4). It follows that the machine $\mathbf{A}'$ recognising L' is

$$\begin{tikzcd}
{} \arrow[r] & 0 \arrow[r, "{a,b}"] & 1 \arrow[d, "{a,b}"] \\
& 3 \arrow[u, "{a,b}"] & 2 \arrow[l, "{a,b}"]
\end{tikzcd}$$

We can see that this was obtained from $\mathbf{A}$ by interchanging terminal and non-terminal states. □

The above example turns out to be typical.

Proposition 2.6.2 *If L is recognised by $\mathbf{A} = (S, A, i, \delta, T)$ then L' is recognised by $\mathbf{A}' = (S, A, i, \delta, T')$ where $T' = S \setminus T$.*

Proof The automaton $\mathbf{A}'$ is exactly the same as the automaton $\mathbf{A}$ *except* that the terminal states of $\mathbf{A}'$ are the non-terminal states of $\mathbf{A}$. We claim that $L(\mathbf{A}') = L'$. To see this we argue as follows. By definition $x \in L(\mathbf{A}')$ if and only if $i \cdot x \in S \setminus T$, which is equivalent to $i \cdot x \notin T$, which means precisely that $x \notin L(\mathbf{A})$. This proves the claim. □

Example 2.6.3 A language $L \subseteq A^*$ is said to be *cofinite* if L' is finite. We proved in Proposition 2.3.4 that every finite language is recognisable. It follows by Proposition 2.6.2 that every cofinite language is recognisable. This example is hopefully an antidote to the mistaken view that people sometimes get when first introduced to finite automata: **the languages recognised by automata need not be finite!** □

The following example motivates our next construction using Boolean operations.

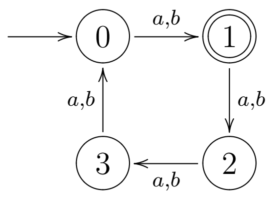
Example 2.6.4 Consider the language

$$N = \{x \in (a+b)^* \colon x \in (a+b)^* aba (a+b)^* \text{ and } |x| \equiv 1 \pmod 4\}.$$

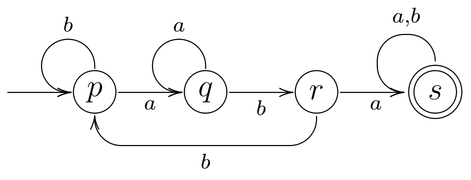
If we define

$$L = \{x \in (a+b)^* \colon |x| \equiv 1 \pmod 4\} \text{ and } M = (a+b)^* aba (a+b)^*,$$

then $N = L \cap M$. Automata that recognise L and M, respectively, are



and



We would like to combine these two automata to build an automaton recognising $N = L \cap M$. To discover how to do this, we need only reflect on how we would decide if a string x is in N: we would run it on the left-hand machine and on the right-hand machine, and we would accept it if and only if when it had been read, both left- and right-hand machines were in a terminal state. To do this, we could run x first on one machine and then on the other, but we could also run it on both machines at the same time. Thus x is input to the left-hand machine in state 0, and a copy on the right-hand machine in state p. The subsequent states of both machines can be recorded by an ordered pair (l, r) where l is the current state of the left-hand machine and r is the current state of the right-hand machine. For example, *abba* causes the two machines to run through the following pairs of states:

$$(0, p), (1, q), (2, r), (3, p), (0, q).$$

The string *abba* is not accepted because although 0 is a terminal state in the left-hand machine, q is not a terminal state in the right-hand machine. □

The above example illustrates the idea behind the following result.

Proposition 2.6.5 *If L and M are recognisable languages over A then so is $L \cap M$.*

Proof Let $L = L(\mathbf{A})$ and $M = L(\mathbf{B})$ where $\mathbf{A} = (S, A, s_0, \delta, F)$ and $\mathbf{B} = (T, A, t_0, \gamma, G)$. Put

$$\mathbf{A} \times \mathbf{B} = (S \times T, A, (s_0, t_0), \delta \times \gamma, F \times G),$$

where

$$(\delta \times \gamma)((s,t), a) = (\delta(s,a), \gamma(t,a));$$

we write

$$(\delta \times \gamma)((s,t), a) = (s,t) \cdot a = (s \cdot a, t \cdot a),$$

as usual. It is easy to check that if x is a string, then the extended transition function has the form

$$(s,t) \cdot x = (s \cdot x, t \cdot x).$$

We claim that $L(\mathbf{A} \times \mathbf{B}) = L \cap M$. By definition $x \in L(\mathbf{A} \times \mathbf{B})$ if and only if $(s_0, t_0) \cdot x \in F \times G$. But this is equivalent to $s_0 \cdot x \in F$ and $t_0 \cdot x \in G$, which says precisely that $x \in L(\mathbf{A}) \cap L(\mathbf{B}) = L \cap M$, and so the claim is proved.$\square$

We have dealt with complementation and intersection, so it is natural to finish off with union. The idea is similar to Proposition 2.6.5.

Proposition 2.6.6 *If L and M are recognisable languages over A then so is $L + M$.*

Proof Let $L = L(\mathbf{A})$ and $M = L(\mathbf{B})$, where $\mathbf{A} = (S, A, s_0, \delta, F)$ and $\mathbf{B} = (T, A, t_0, \gamma, G)$. Put

$$\mathbf{A} \sqcup \mathbf{B} = (S \times T, A, (s_0, t_0), \delta \times \gamma, (F \times T) + (S \times G)).$$

We claim that $L(\mathbf{A} \sqcup \mathbf{B}) = L + M$. By definition $x \in L(\mathbf{A} \sqcup \mathbf{B})$ if and only if $(s_0, t_0) \cdot x \in (F \times T) + (S \times G)$. This is equivalent to $s_0 \cdot x \in F$ *or* $t_0 \cdot x \in G$, because $s_0 \cdot x \in S$ *and* $t_0 \cdot x \in T$ always hold. Hence $x \in L(\mathbf{A}) + L(\mathbf{B}) = L + M$, and the claim is proved. $\square$

There is an alternative proof of Proposition 2.6.6, which relies only on Propositions 2.6.2 and 2.6.5 together with a little set theory. See Exercises 2.6.

Observe that the only difference between automata constructed in Proposition 2.6.5 and Proposition 2.6.6 lies in the definition of the terminal states: to recognise the *intersection* of two languages the terminal states are those ordered pairs (s,t) where s *and* t are terminal; to recognise the *union* of two languages the terminal states are those ordered pairs (s,t) where s *or* t is terminal.

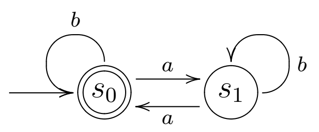
Example 2.6.7 Let $A = \{a,b\}$. We wish to construct an automaton to recognise the language

$$L = \{x \in A^*\colon\ |\,x\,|_a \text{ is even and } |\,x\,|_b \text{ is odd}\}.$$

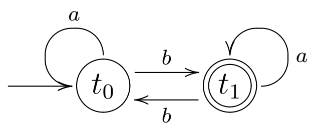
This language is the intersection of

$$M = \{x \in A^*\colon\ |\,x\,|_a \text{ is even}\} \text{ and } N = \{x \in A^*\colon\ |\,x\,|_b \text{ is odd}\}.$$

It is easy to construct automata that recognise these languages separately; the machine **A** below recognises M:



and the machine **B** below recognises N:



To construct the machine $\mathbf{A} \times \mathbf{B}$ (and similar comments apply to the construction of $\mathbf{A} \sqcup \mathbf{B}$) we proceed as follows. The set of states of $\mathbf{A} \times \mathbf{B}$ is the set $S \times T$, where S is the set of states of **A** and T is the set of states of **B**. In this case,

$$S \times T = \{(s_0,t_0),(s_0,t_1),(s_1,t_0),(s_1,t_1)\}.$$

We draw and label these four states. Mark the initial state, which is (s_0,t_0). Mark the set of terminal states, which in this case is just (s_0,t_1); it is only at this point that the constructions of $\mathbf{A} \times \mathbf{B}$ and $\mathbf{A} \sqcup \mathbf{B}$ differ. It remains now to insert all the transitions. For each $a \in A$ and each pair $(s,t) \in S \times T$, calculate $(s,t)\cdot a$ which by definition is just $(s\cdot a, t\cdot a)$. For example,

$$(s_0,t_0)\cdot a = (s_0\cdot a, t_0\cdot a) = (s_1,t_0)$$

and

$$(s_0, t_0) \cdot b = (s_0 \cdot b, t_0 \cdot b) = (s_0, t_1).$$

We therefore draw an arrow labelled a from the state labelled (s_0, t_0) to the state labelled (s_1, t_0), and an arrow labelled b from the state labelled (s_0, t_0) to the state labelled (s_0, t_1). Continuing in this way, the machine $\mathbf{A} \times \mathbf{B}$, that recognises the language $L = M \cap N$, has the following form:

$$\begin{tikzcd}
{} \arrow[r] & s_0, t_0 \arrow[r, shift left, "a"] \arrow[d, shift left, "b"] & s_1, t_0 \arrow[l, shift left, "a"] \arrow[d, shift left, "b"] \\
& \circledcirc\, s_0, t_1 \arrow[r, shift left, "a"] \arrow[u, shift left, "b"] & s_1, t_1 \arrow[l, shift left, "a"] \arrow[u, shift left, "b"]
\end{tikzcd}$$

□

Exercises 2.6

1. Construct separate automata to recognise each of the languages below:

$$L = \{w \in \{0,1\}^* \colon |\, w\,|_0 \equiv 1 \pmod 3\}$$

and

$$M = \{w \in \{0,1\}^* \colon |\, w\,|_1 \equiv 2 \pmod 3\}.$$

Use Proposition 2.6.5 to construct an automaton that recognises $L \cap M$.

2. Let

$$L = \{x \in a^* \colon |x| \equiv 0 \pmod 3\} \text{ and } M = \{x \in a^* \colon |x| \equiv 0 \pmod 5\}.$$

Construct automata that recognise L and M, respectively. Use Proposition 2.6.6 to construct an automaton that recognises $L + M$.

3. Prove that if L and M are recognisable languages over A, then so is $L \setminus M$.

4. Show how the constructions of $\mathbf{A}'$ and $\mathbf{A} \times \mathbf{B}$ combined with one of de Morgan's laws enables $\mathbf{A} \sqcup \mathbf{B}$ to be constructed.

5. Show that if $L_1, \ldots, L_n$ are each recognisable, then so too are $L_1 + \ldots + L_n$ and $L_1 \cap \ldots \cap L_n$.

6. Let $L = \{x \in (a+b)^*\colon |x|_a = |x|_b\}$. Show that L is not recognisable.

2.7 Summary of Chapter 2

- *Incomplete automata*: An automaton is incomplete if there are missing transitions. An incomplete automaton $\mathbf{A}$ can easily be converted into a complete automaton $\mathbf{A}^c$ recognising the same language by simply adding a sink state: this is a state to which missing transitions are connected but from which there is no escape.

- *Automata over one letter alphabets*: These are described by 'saucepan automata'.

- *Automata that count*: By arranging states in a circle it is possible to count modulo n. Automata can also be constructed to count relative to a threshold by arranging the states in a line.

- *Automata that locate patterns:* Automata can be explicitly constructed to recognise the languages xA^*, A^*xA^*, and A^*x where A is any alphabet and x is any non-empty string.

- *Recognising Boolean combinations of languages*: If $L = L(\mathbf{A})$ and $M = L(\mathbf{B})$, then there are algorithms for combining $\mathbf{A}$ and $\mathbf{B}$ to recognise $L + M$ and $L \cap M$. This is also an algorithm to convert $\mathbf{A}$ into an automaton recognising L'.

Chapter 3

Non-deterministic automata

In Chapter 2, we looked at various ways of constructing an automaton to recognise a given language. However, we did not get very far. The reason was that automata as we have defined them are quite 'rigid': from each state we must have exactly one transition labelled by each element of the input alphabet. This is a strong constraint and restricts our freedom considerably when designing an automaton. To make progress, we need a tool that is easy to use and can help us design automata. This is the role played by the non-deterministic automata we introduce in this chapter. A non-deterministic automaton is exactly like an automaton except that we allow multiple initial states and we impose no restrictions on transitions as long as they are labelled by symbols in the input alphabet. Using such automata, it is often easy, as we shall see, to construct a non-deterministic automaton recognising a given language. However, this would be useless unless we had some way of converting a non-deterministic automaton into a deterministic one recognising the same language. We describe an algorithm that does exactly this. We can therefore add non-deterministic automata to our toolbox for building automata.

3.1 Accessible automata

There are many different automata that can be constructed to recognise a given language. All things being equal, we would like an automaton with the smallest number of states. In Chapter 6, we will investigate this problem in detail. For the time being, we shall look at one technique that may make an automaton smaller without changing the language it recognises, and that will play an important role in our algorithm for converting a non-deterministic automaton into a deterministic automaton in Section 3.2.

Let $\mathbf{A} = (S, A, i, \delta, T)$ be a finite automaton. We say that a state $s \in S$ is *accessible* if there is a string $x \in A^*$ such that $i \cdot x = s$. A state that is not accessible is said to be *inaccessible*. An automaton is said to be *accessible* if every state is accessible. In an accessible automaton, each state can be reached from the initial state by means of some input string. Observe that the initial state itself is always accessible because $i \cdot \varepsilon = i$. It is clear that the inaccessible states of an automaton can play no role in accepting strings, consequently we expect that they could be removed without the language being changed. This turns out to be the case as we now show.

Let $\mathbf{A} = (S, A, i, \delta, T)$ be a finite automaton. Define a new machine,

$$\mathbf{A}^a = (S^a, A, i^a, \delta^a, T^a),$$

as follows:

- S^a is the set of accessible states in S.
- $i^a = i$.
- $T^a = T \cap S^a$, the set of accessible terminal states.
- δ^a has domain $S^a \times A$, codomain S^a but otherwise behaves like δ.

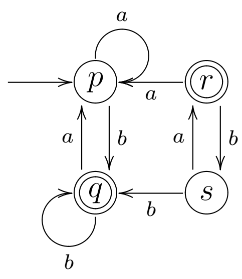
The way $\mathbf{A}^a$ is constructed from $\mathbf{A}$ can be put very simply: erase all inaccessible states from $\mathbf{A}$ and all transitions that either start or end at an inaccessible state.

Proposition 3.1.1 *Let $\mathbf{A} = (S, A, i, \delta, T)$ be a finite automaton. Then $\mathbf{A}^a$ is an accessible automaton and $L(\mathbf{A}^a) = L(\mathbf{A})$.*

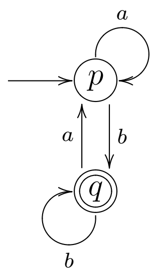
Proof It is clear that $\mathbf{A}^a$ is a well-defined accessible automaton. It is also obvious that $L(\mathbf{A}^a) \subseteq L(\mathbf{A})$. To show that $L(\mathbf{A}^a) = L(\mathbf{A})$, it only remains to prove that $L(\mathbf{A}) \subseteq L(\mathbf{A}^a)$. Let $x \in L(\mathbf{A})$. Then $i \cdot x \in T$, and every state in the path labelled by x from i to $i \cdot x$ is accessible. Thus this path also lies in $\mathbf{A}^a$. It follows that $x \in L(\mathbf{A}^a)$, as required. $\square$

The automaton $\mathbf{A}^a$ is called the *accessible part* of $\mathbf{A}$. When the number of states is small, it is easy to construct $\mathbf{A}^a$.

Example 3.1.2 Let **A** be the automaton below:



It is clear that p and q are both accessible since $p = p \cdot \varepsilon$ and $q = p \cdot b$, and that neither r nor s are accessible. Thus in this case, $\mathbf{A}^a$ is the following:



This machine is obtained by erasing the non-accessible states r and s and all transitions to and from these two states. □

However, when there are many states, the construction of $\mathbf{A}^a$ is not quite so straightforward. The following lemma lays the foundations for an algorithm for constructing $\mathbf{A}^a$. It says that if a state is accessible, then it can be reached by a string whose length is strictly less than the number of states in the machine.

Lemma 3.1.3 *Let* $\mathbf{A} = (S, A, s_0, \delta, T)$ *be a finite automaton with set of states* S*. If* s *is an accessible state, then there exists a string* $x \in A^*$ *such that* $|\,x\,| < |\,S\,|$ *and* $s_0 \cdot x = s$*.*

Proof Let $s \in S$ be an accessible state. By definition there exists $x \in A^*$ such that $s_0 \cdot x = s$. Let $x \in A^*$ be a string of *smallest possible length* such that $s_0 \cdot x = s$. We would like to show that $|\,x\,| < |\,S\,|$, so for the sake of argument, assume instead that $|\,x\,| \geq |\,S\,|$. Let $x = x_1 \ldots x_n$ where $x_i \in A$ and $n = |\,x\,| \geq |\,S\,|$. Consider the sequence of states,

$$s_0,\ s_1 = s_0 \cdot x_1,\ s_2 = s_0 \cdot (x_1 x_2), \ldots,\ s_n = s_0 \cdot (x_1 \ldots x_n).$$

Since $n \geq |\,S\,|$ it follows that $n+1 > |\,S\,|$. But $s_0, s_1, \ldots, s_n$ is a list of states of length $n+1$ so there must be some repetition of states in this list. Let $i \neq j$ be subscripts such that $s_i = s_j$. We have the following schematic diagram of the path in $\mathbf{A}$ labelled by the string x:

$$\begin{tikzcd}
{} \arrow[r] & s_0 \arrow[r, "x_1 \ldots x_i"] & s_i \arrow[r, "x_{j+1} \ldots x_n"] \arrow[loop below, "x_{i+1} \ldots x_j"] & \boxed{s_n}
\end{tikzcd}$$

Put $x' = x_1 \ldots x_i x_{j+1} \ldots x_n$; in other words, cut out the factor of x which labels the loop. Then $|\,x'\,| < |\,x\,|$ and $s_0 \cdot x' = s$, which contradicts our choice of x. Consequently, we must have $|\,x\,| < |\,S\,|$. □

The above result implies that we can find S^a in the following way. Let $n = |\,S\,|$ and denote the initial state by s_0. If $X \subseteq S$ and $L \subseteq A^*$ then define

$$X \cdot L = \{x \cdot a \colon x \in X \text{ and } a \in L\}.$$

The set of strings over A of length at most $n-1$ is just

$$\sum_{i=0}^{n-1} A^i = A^0 + \ldots + A^{n-1}.$$

Thus Lemma 3.1.3 can be expressed in the following way:

$$S^a = s_0 \cdot \left(\sum_{i=0}^{n-1} A^i \right) = \sum_{i=0}^{n-1} s_0 \cdot A^i.$$

To calculate the terms in this union, we need only calculate in turn the sets,

$$S_0 = \{s_0\}, \quad S_1 = S_0 \cdot A, \quad S_2 = S_1 \cdot A, \ldots, S_{n-1} = S_{n-2} \cdot A,$$

because $S_j = s_0 \cdot A^j$.

These calculations can be very easily put into the form of a sequence of rooted trees. By the ‘distance’ between two vertices in a tree we mean the length of the shortest path joining the two vertices. The ‘height’ of the tree is the length of the longest path from root to leaf with no repeated vertices. The root of the tree is labelled s_0. For each $a \in A$ construct an arrow from s_0 to $s_0 \cdot a$. In general, if s is the label of a vertex, then draw arrows from

s to $s \cdot a$ for each $a \in A$. The vertices at the distance j from the root are precisely the elements of $s_0 \cdot A^j$. Thus the process will terminate when the tree has height $n - 1$. The vertices of this tree are precisely the accessible states of the automaton. The automaton $\mathbf{A}^a$ can now be constructed from $\mathbf{A}$ by erasing all non-accessible vertices and the transitions that go to or from them.

The drawback of this algorithm is that if the automaton has n states, then all of the tree to height $n-1$ has to be drawn. However such a tree contains repeated information: a state can appear more than once and, where it is repeated, no *new* information will be constructed from it.

The following construction omits the calculation of $s \cdot A$ whenever s is a repeat, which means that the whole tree is not constructed; in addition, it also enables us to detect when all accessible states have been found without having to count.

Algorithm 3.1.4 (Transition tree of an automaton) Let $\mathbf{A}$ be an automaton. The *transition tree* of $\mathbf{A}$ is constructed inductively in the following way. We assume that a linear ordering of A is specified at the outset so we can refer meaningfully to 'the elements of A in turn':

(1) The root of the tree is s_0 and we put $T_0 = \{s_0\}$.

(2) Assume that T_i has been constructed; vertices in T_i will have been labelled either 'closed' or 'non-closed.' The meaning of these two terms will be made clear below. We now show how to construct T_{i+1}.

(3) For each non-closed leaf s in T_i and for each $a \in A$ in turn construct an arrow from s to $s \cdot a$ labelled by a; if, in addition, $s \cdot a$ is a repeat of any state that has already been constructed, then we say it is *closed* and mark it with a $\times$.

(4) The algorithm terminates when all leaves are closed.

$\square$

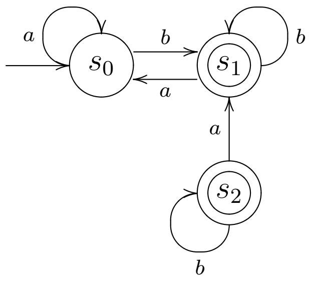
We have to prove that the algorithm above is what we want.

Proposition 3.1.5 *Let $|S| = n$. Then there exists an $m \leq n$ such that every leaf in T_m is closed, and S^a is just the set of vertices in T_m.*

Proof Let $s \in S^a$ and let $x \in A^*$ be the smallest string in the tree order such that $s_0 \cdot x = s$. Then s first appears as a vertex in the tree $T_{|x|}$. By Lemma 3.1.3, the latest an accessible state can appear for the first time is in T_{n-1}. Thus at worst all states in T_n are closed. $\square$

The transition tree not only tells us the accessible states of $\mathbf{A}$ but can also be used to construct $\mathbf{A}^a$ as follows: erase the $\times$'s and glue leaves to interior vertices with the same label. The diagram that results is the transition diagram of $\mathbf{A}^a$. All of this is best illustrated by means of an example.

Example 3.1.6 Consider the automaton $\mathbf{A}$ pictured below:



We shall step through the algorithm for constructing the transition tree of $\mathbf{A}$ and so construct $\mathbf{A}^a$. Of course, it is easy to construct $\mathbf{A}^a$ directly in this case, but it is the algorithm we wish to illustrate.

The tree T_0 is just

$$s_0$$

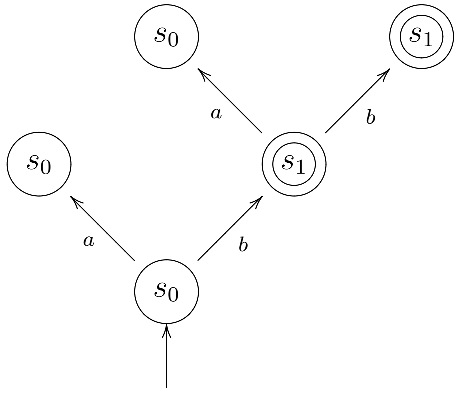
The tree T_1 is

$$\begin{tikzcd}
s_0, \times & & s_1 \\
& s_0 \arrow[ul, "a"'] \arrow[ur, "b"'] &
\end{tikzcd}$$

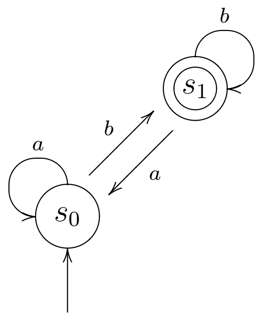
The tree T_2 is

$$\begin{tikzcd}
& s_0, \times & & s_1, \times \\
s_0, \times & & s_1 \arrow[ul, "a"'] \arrow[ur, "b"'] & \\
& s_0 \arrow[ul, "a"'] \arrow[ur, "b"'] & &
\end{tikzcd}$$

T_2 is the transition tree because every leaf is closed. This tree can be transformed into an automaton as follows. Erase all $\times$'s, and mark initial and terminal states. This gives us the following:



Now glue the leaves to the interior vertices labelling the same state. We obtain the following automaton:

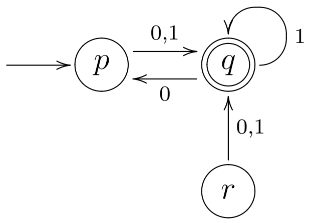


This is the automaton $\mathbf{A}^a$. □

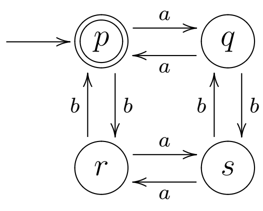
Exercises 3.1

1. Construct transition trees for the automata below.

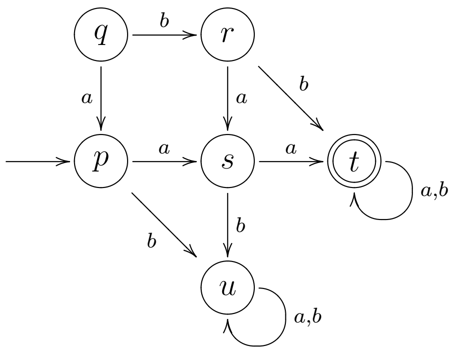
(i)



(ii)



(iii)



3.2 Non-deterministic automata

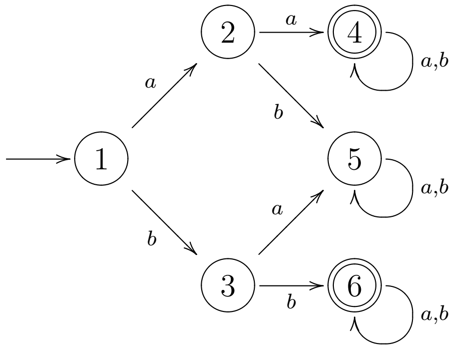
Deterministic automata are intended to be models of real machines. The non-deterministic automata we introduce in this section should be regarded as tools helpful in designing deterministic automata rather than as models of real-life machines. To motivate the definition of non-deterministic automata, we shall consider a simple problem.

Let A be an alphabet. If $x = a_1 \dots a_n$ where $a_i \in A$, then the *reverse of* x, denoted $\mathrm{rev}(x)$, is the string $a_n \dots a_1$. We define $\mathrm{rev}(\varepsilon) = \varepsilon$. Clearly, $\mathrm{rev}(\mathrm{rev}(x)) = x$, and $\mathrm{rev}(xy) = \mathrm{rev}(y)\mathrm{rev}(x)$. If L is a language then the *reverse of* L, denoted by $\mathrm{rev}(L)$, is the language

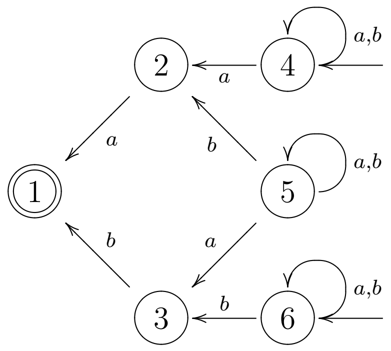
$$\mathrm{rev}(L) = \{\mathrm{rev}(x) \colon x \in L\}.$$

Consider now the following question: if L is recognisable, then is $\mathrm{rev}(L)$ recognisable? To see what might be involved in answering this problem, we consider an example.

Example 3.2.1 Let $L = (aa + bb)(a + b)^*$, the language of all strings of a's and b's that begin with a double letter. This language is recognised by the following automaton:



In this case, rev(L) is the language of all strings of a's and b's that *end* with a double letter. In order to construct an automaton to recognise this language, it is tempting to modify the above automaton in the following way: reverse all the transitions, and interchange initial and terminal states, like this:



This diagram violates the rules for the transition diagram of a complete, deterministic finite-state automaton in two fundamental ways: there is more than one initial state, and there are forbidden configurations. However, if we put to one side these fatal problems, we do notice an interesting property of this diagram: the strings that label paths in the diagram, which begin at one of the initial states and conclude at the terminal state, form precisely the language rev(L): those strings that *end* in a double letter. □

This diagram is in fact an example of a non-deterministic automaton. After giving the formal definition below, we shall prove that every non-deterministic automaton can be converted into a deterministic automaton recognising the same language. An immediate application of this result can be obtained by generalising the example above; we will therefore be able to prove that the reverse of a recognisable language is also recognisable.

Recall that if X is a set, then $\mathsf{P}(X)$ is the power set of X, the set of all subsets of X. The set $\mathsf{P}(X)$ contains both $\emptyset$ and X as elements. A *non-deterministic automaton* $\mathbf{A}$ is determined by five pieces of information:

$$\mathbf{A} = (S, A, I, \delta, T),$$

where S is a finite set of states, A is the input alphabet, I is a set of initial states, δ: $S \times A \to \mathsf{P}(S)$ is the transition function, and T is a set of terminal states.

In addition to allowing any number of initial states, the key feature of this definition is that $\delta(s, a)$ is now a subset of S (possibly empty!). We can draw transition diagrams and transition tables just as we did for deterministic machines. The transition table of the machine we constructed in Example 3.2.1 is as follows:

	a	b
$\leftarrow 1$	$\emptyset$	$\emptyset$
2	$\{1\}$	$\emptyset$
3	$\emptyset$	$\{1\}$
$\rightarrow 4$	$\{2,4\}$	$\{4\}$
5	$\{3,5\}$	$\{2,5\}$
$\rightarrow 6$	$\{6\}$	$\{3,6\}$

It now remains to define the analogue of the 'extended transition function.' For each string $x \in A$ and state $s \in S$ we want to know *the set of all possible states* that can be reached by paths starting at s and labelled by x. The formal definition is as follows and, as in the deterministic case, the function being defined exists and is unique. The function δ^* is the unique function from $S \times A^*$ to $\mathsf{P}(S)$ satisfying the following three conditions where $a \in A$, $w \in A^*$ and $s \in S$:

(ETF1) $\delta^*(s, \varepsilon) = \{s\}$.

(ETF2) $\delta^*(s, a) = \delta(s, a)$.

(ETF3) $\delta^*(s, aw) = \sum_{q \in \delta(s,a)} \delta^*(q, w)$.

Condition (ETF3) needs a little explaining. Suppose that $\delta(s,a) = \{q_1, \ldots, q_n\}$. Then condition (ETF3) means

$$\delta^*(q_1, w) + \ldots + \delta^*(q_n, w).$$

Notation We shall usually write $s \cdot x$ rather than $\delta^*(s,x)$, but it is important to remember that $s \cdot x$ is a *set* in this case.

Let us check that this definition captures what we intended.

Lemma 3.2.2 *Let $x = a_1 \ldots a_n \in A^*$. Then $t \in s \cdot x$ if and only if there exist states $q_1, \ldots, q_n = t$ such that $q_1 \in s \cdot a_1$, and $q_i \in q_{i-1} \cdot a_i$ for $i = 2, \ldots, n$.*

Proof The condition simply says that $t \in s \cdot x$ if and only if the string x labels some path in $\mathbf{A}$ starting at s and ending at t. Observe that $t \in \delta^*(s, ax)$ if and only if $t \in \delta^*(q, x)$ for some state $q \in \delta(s,a)$. By applying this observation repeatedly, starting with $ax = a_1 \ldots a_n$, we obtain the desired result.$\square$

The language $L(\mathbf{A})$ is defined to be

$$L(\mathbf{A}) = \{w \in A^* \colon \left(\sum_{q \in I} q \cdot w\right) \cap T \neq \emptyset\}.$$

That is, the language recognised by a non-deterministic automaton consists of all strings that label paths starting at one of the initial states and ending at one of the terminal states.

It might be thought that because there is a degree of choice available, non-deterministic automata might be more powerful than deterministic automata, meaning that non-deterministic automata might be able to recognise languages that deterministic automata could not. In fact, this is not so. To prove this, we shall make use of the following construction. Let $\mathbf{A} = (S, A, I, \delta, T)$ be a non-deterministic automaton. We construct a deterministic automaton $\mathbf{A}^d = (S^d, A, i^d, \Delta, T^d)$ as follows:

- $S^d = \mathsf{P}(S)$; the set of states is labelled by the subsets of S.
- $i^d = I$; the initial state is labelled by the subset consisting of all the initial states.

- $T^d = \{Q \in \mathsf{P}(S) \colon Q \cap T \neq \emptyset\}$; the terminal states are labelled by the subsets that contain at least one terminal state.
- For $a \in A$ and $Q \in \mathsf{P}(S)$ define
$$\Delta(Q, a) = \sum_{q \in Q} q \cdot a;$$
this means that the subset $\Delta(Q, a)$ consists of all states in S that can be reached from states in Q by following a transition labelled by a.

It is clear that $\mathbf{A}^d$ is a complete, deterministic, finite automaton.

Theorem 3.2.3 (Subset construction) *Let* $\mathbf{A}$ *be a non-deterministic automaton. Then* $\mathbf{A}^d$ *is a deterministic automaton such that* $L(\mathbf{A}^d) = L(\mathbf{A})$.

Proof The main plank of the proof will be to relate the extended transition function Δ^* in the deterministic machine $\mathbf{A}^d$ to the extended transition function δ^* in the non-deterministic machine $\mathbf{A}$. We shall prove that for any $Q \subseteq S$ and $x \in A^*$ we have that
$$\Delta^*(Q, x) \;=\; \sum_{q \in Q} \delta^*(q, x). \tag{3.1}$$
This is most naturally proved by induction on the length of the string x.

For the base step, we prove the theorem holds when $x = \varepsilon$. By the definition of Δ, we have that $\Delta^*(Q, \varepsilon) = Q$, whereas by the definition of δ^* we have that $\sum_{q \in Q} \delta^*(q, \varepsilon) = \sum_{q \in Q} \{q\} = Q$.

For the induction hypothesis, assume that (3.1) holds for all strings $x \in A^*$ satisfying $|\,x\,| = n$. Consider now the string y where $|\,y\,| = n+1$. We can write $y = ax$ where $a \in A$ and $x \in A^*$ and $|\,x\,| = n$. From the definition of Δ^* we have
$$\Delta^*(Q, y) = \Delta^*(Q, ax) = \Delta^*(\Delta(Q, a), x).$$
Put $Q' = \Delta(Q, a)$. Then
$$\Delta^*(Q, y) = \Delta^*(Q', x) = \sum_{q' \in Q'} \delta^*(q', x)$$
by the induction hypothesis. From the definitions of Q' and Δ we have that
$$\sum_{q' \in Q'} \delta^*(q', x) = \sum_{q \in Q} \left(\sum_{q' \in \delta(q,a)} \delta^*(q', x) \right).$$

By the definition of δ^* we have that

$$\sum_{q \in Q} \left(\sum_{q' \in \delta(q,a)} \delta^*(q', x) \right) = \sum_{q \in Q} \delta^*(q, ax) = \sum_{q \in Q} \delta^*(q, y).$$

This proves the claim.

We can now easily prove that $L(\mathbf{A}) = L(\mathbf{A}^d)$. By definition,

$$x \in L(\mathbf{A}) \Leftrightarrow \left(\sum_{q \in I} \delta^*(q, x) \right) \cap T \neq \emptyset.$$

From the definition of the terminal states in $\mathbf{A}^d$ this is equivalent to

$$\sum_{q \in I} \delta^*(q, x) \in T^d.$$

We now use equation (3.1) to obtain

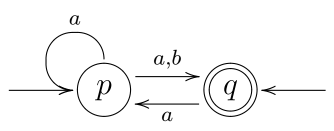
$$\Delta^*(I, x) \in T^d.$$

This is of course equivalent to $x \in L(\mathbf{A}^d)$. □

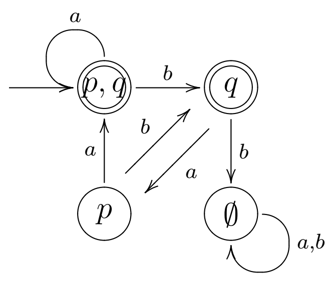
The automaton $\mathbf{A}^d$ is called the *determinised* version of $\mathbf{A}$.

Notation Let $\mathbf{A}$ be a non-deterministic automaton with input alphabet A. Let Q be a set of states and $a \in A$. We denote by $Q \cdot a$ the set of all states that can be reached by starting in Q and following transitions labelled only by a. In other words, $\Delta(Q, a)$ in the above theorem. We define $Q \cdot A = \sum_{a \in A} Q \cdot a$.

Example 3.2.4 Consider the following non-deterministic automaton $\mathbf{A}$:



The corresponding deterministic automaton constructed according to the subset construction has 4 states labelled $\emptyset$, $\{p\}$, $\{q\}$, and $\{p, q\}$; the initial state is the state labelled $\{p, q\}$ because this is the set of initial states of $\mathbf{A}$; the terminal states are $\{q\}$ and $\{p, q\}$ because these are the only subsets of $\{p, q\}$ that contain terminal states of $\mathbf{A}$; the transitions are calculated according to the definition of Δ. Thus $\mathbf{A}^d$ is the automaton:



Observe that the state labelled by the empty set is a sink state, as it always must be. □

The obvious drawback of the subset construction is the huge increase in the number of states in passing from $\mathbf{A}$ to $\mathbf{A}^d$. Indeed, if $\mathbf{A}$ has n states, then $\mathbf{A}^d$ will have 2^n states. This is sometimes unavoidable as we ask you to show in Exercises 3.2, but often the machine $\mathbf{A}^d$ will contain many inaccessible states. There is an easy way of avoiding this: construct the transition tree of $\mathbf{A}^d$ directly from $\mathbf{A}$ and so construct $(\mathbf{A}^d)^a = \mathbf{A}^{da}$. This is done as follows.

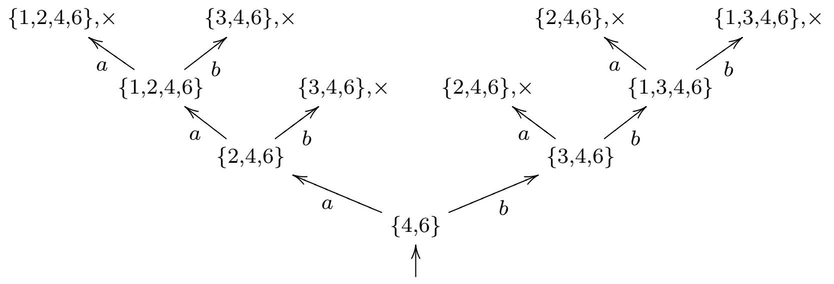
Algorithm 3.2.5 (Accessible subset construction) The input to this algorithm is a non-deterministic automaton $\mathbf{A} = (S, A, I, \delta, T)$ and the output is $\mathbf{A}^{da}$, an accessible deterministic automaton recognising $L(\mathbf{A})$. The procedure is to construct the transition tree of $\mathbf{A}^d$ directly from $\mathbf{A}$. The root of the tree is the set I. Apply the algorithm for the transition tree by constructing as vertices $Q \cdot a$ for each non-closed vertex Q and input letter a. □

We show how this algorithm works by applying it to the non-deterministic automaton constructed in Example 3.2.1.

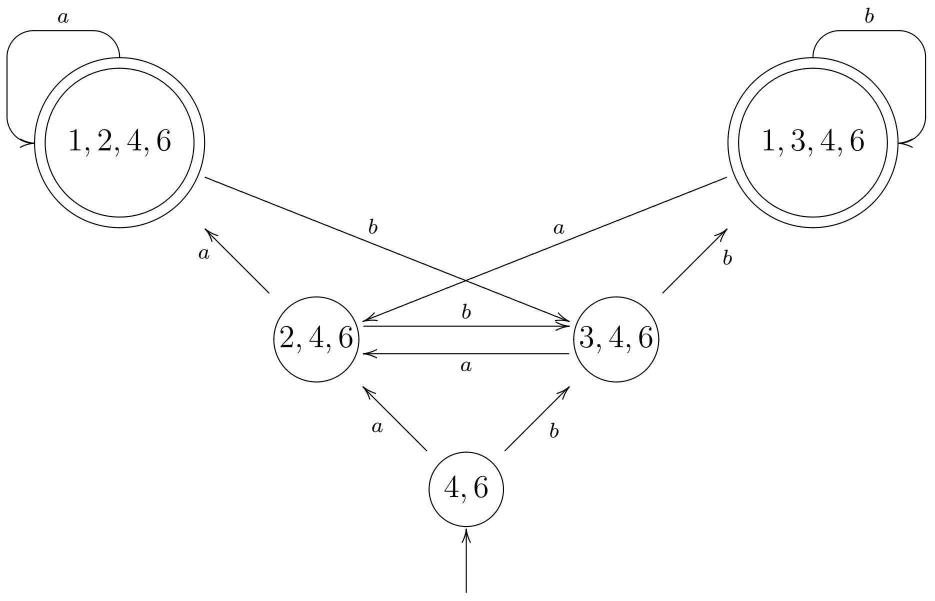
Example 3.2.6 The root of the tree is labelled $\{4, 6\}$, the *set* of initial states of the non-deterministic automaton. The next step in the algorithm yields the tree:

$$\begin{tikzcd}
\{2,4,6\} & & \{3,4,6\} \\
& \{4,6\} \arrow[ul, "a"] \arrow[ur, "b"'] & \\
& {} \arrow[u] &
\end{tikzcd}$$

Continuing with the algorithm, we obtain the transition tree of $(\mathbf{A}^d)^a = \mathbf{A}^{da}$:



Finally, we obtain the automaton $\mathbf{A}^{da}$ pictured below:

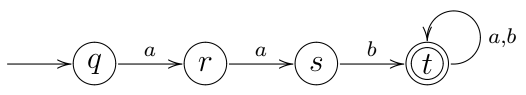


□

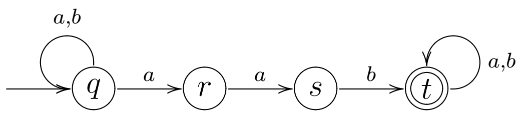
Exercises 3.2

1. Apply the accessible subset construction to the non-deterministic automata below. Hence find deterministic automata which recognise the same language in each case.

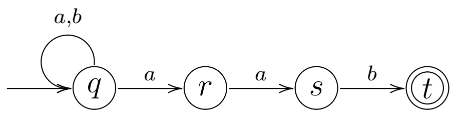
(i)



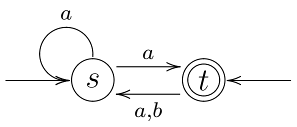
(ii)



(iii)



(iv)



2. Find a non-deterministic automaton with 4 states that recognises the language $(0 + 1)^*1(0 + 1)^2$. Use the accessible subset construction to find a deterministic automaton that recognises the same language.

3. Let $n \geq 1$. Show that the language $(0+1)^*1(0+1)^{n-1}$ can be recognised by a non-deterministic automaton with $n+1$ states. Show that any deterministic automaton that recognises this language must have at least 2^n states.

This example shows that an exponential increase in the number of states in passing from a non-deterministic automaton to a corresponding deterministic automaton is sometimes unavoidable.

3.3 Applications

Non-deterministic automata make designing certain kinds of automata easy: we may often simply write down a non-deterministic automaton and then apply the accessible subset construction. It is however worth pointing out that the automata obtained by applying the accessible subset construction will often have some rather obvious redundancies and can be simplified further. The general procedure for doing this will be described in Chapter 6.

In this section, we look at some applications of non-deterministic automata. Our first result generalises Example 3.2.1.

Proposition 3.3.1 *Let L be recognisable. Then* $\mathrm{rev}(L)$*, the reverse of L, is recognisable.*

Proof Let $L = L(\mathbf{A})$, where $\mathbf{A} = (S, A, I, \delta, T)$ is a non-deterministic automaton. Define another non-deterministic automaton $\mathbf{A}^{\text{rev}}$ as follows:

$$\mathbf{A}^{\text{rev}} = (S, A, T, \gamma, I),$$

where γ is defined by $s \in \gamma(t, a)$ if and only if $t \in \delta(s, a)$; in other words, we reverse the arrows of $\mathbf{A}$ and relabel the initial states as terminal and vice versa. It is now straightforward to check that $x \in L(\mathbf{A}^{\text{rev}})$ if and only if $\text{rev}(x) \in L(\mathbf{A})$. Thus $L(\mathbf{A}^{\text{rev}}) = \text{rev}(L)$. □

The automaton $\mathbf{A}^{\text{rev}}$ is called the *reverse* of $\mathbf{A}$.

Non-deterministic automata provide a simple alternative proof to Proposition 2.4.6.

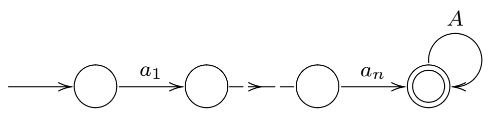
Proposition 3.3.2 *Let L and M be recognisable. Then $L + M$ is recognisable.*

Proof Let $\mathbf{A}$ and $\mathbf{B}$ be non-deterministic automata, recognising L and M, respectively. Lay them side by side; the result is a non-deterministic automaton recognising $L + M$. □

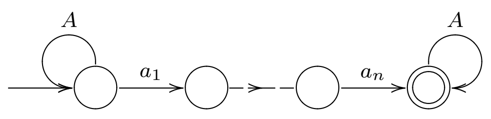
Languages defined in terms of the presence of patterns of various kinds can easily be proved to be recognisable using non-deterministic automata.

Proposition 3.3.3 *Let A be an alphabet and let $w = a_1 \dots a_n$ be a non-empty string over A. Each of the languages wA^*, A^*wA^* and A^*w is recognisable.*

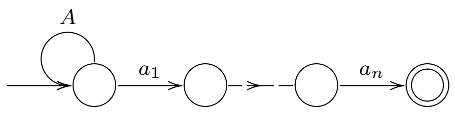
Proof Each language can be recognised by the respective non-deterministic automaton below, which I have drawn schematically:



and



and



Each of these can be converted into an equivalent deterministic automaton using the accessible subset construction. □

For our final application, we can prove that certain Kleene stars are recognisable.

Proposition 3.3.4 *Let X be a finite set of strings none of which is empty. Then X^* is recognisable.*

Proof We construct a non-deterministic automaton that recognises X^*. Draw a single initial state which is also terminal. Call this the base. For each string $x \in X$ we construct a 'petal' attached to the base — the resulting automaton is called a 'petal automaton'. If $x = a_1 \ldots a_r$ where $r \geq 1$, add states $s_1, \ldots, s_{r-1}$ where a_1 labels the transition from the base to s_1, a_2 labels the transition from s_1 to s_2, a_3 labels the transition from s_2 to s_3, and so on with a_r labelling the transition from s_{r-1} back to the base. It is easy to see that the petal automaton recognises X^*. □

Finally, it is easy to construct non-deterministic automata that recognise all strings that contain a specific string as a substring. How this is done we leave to the reader.

Exercises 3.3

1. Construct non-deterministic automata recognising the following languages over the alphabet $A = \{a, b\}$.

(i) $(a^2 + ab + b^2)(a + b)^*$.

(ii) $(a + b)^*(a^2 + ab + b^2)$.

(iii) $(a + b)^*(aaa + bbb)(a + b)^*$.

(iv) $(a^2 + ba + b^2 + ba^2 + b^2a)^*$.

(v) $(a+b)^*a(a+b)^*a(a+b)^*b(a+b)^*$.

2. Construct an incomplete automaton $\mathbf{A} = (S, A, i, \delta, T)$ such that the automaton $\mathbf{B} = (S, A, i, \delta, S \setminus T)$ does not recognise $L(\mathbf{A})'$, the complement of $L(\mathbf{A})$.

It is only possible to prove that the complement of a recognisable language is recognisable using complete deterministic automata.

3.4 Summary of Chapter 3

- *Accessible automata*: A state s is accessible if there is an input string that labels a path starting at the initial state and ending at the state s. An automaton is accessible if every state is accessible. If $\mathbf{A}$ is an automaton, then the accessible part of $\mathbf{A}$, denoted by $\mathbf{A}^a$, is obtained by removing all inaccessible states and transitions to and from them. The language remains unaltered. There is an efficient algorithm for constructing $\mathbf{A}^a$ using the transition tree of $\mathbf{A}$.

- *Non-deterministic automata*: These are automata where the restrictions of completeness and determinism are renounced and where we are allowed to have a set of initial states. A string is accepted by such a machine if it labels at least one path from an initial state to a terminal state. If $\mathbf{A}$ is a non-deterministic automaton, then there is an algorithm, called the subset construction, which constructs a deterministic automaton, denoted $\mathbf{A}^d$, recognising the same language as $\mathbf{A}$. The disadvantage of this construction is that the states of $\mathbf{A}^d$ are labelled by the subsets of the set of states of $\mathbf{A}$. The accessible subset construction constructs the automaton $(\mathbf{A}^d)^a = \mathbf{A}^{da}$ directly from $\mathbf{A}$ using the transition tree and often leads to a much smaller automaton.

- *Applications of non-deterministic automata*: Let $\mathbf{A}$ be a non-deterministic automaton. Then $\mathbf{A}^{\text{rev}}$, the reverse of $\mathbf{A}$, is obtained by reversing all the transitions in $\mathbf{A}$ and interchanging initial and terminal states. The language recognised by $\mathbf{A}^{\text{rev}}$ is the reverse of the language recognised by $\mathbf{A}$. If L and M are recognisable, then we can prove that $L + M$ is recognisable using non-deterministic automata. If A is an alphabet and w a non-empty string over A then the following languages are all recognisable: wA^*, A^*wA^*, and A^*w. If X is a finite set of non-empty strings then the petal automaton of X recognises X^*. Finally the set of all strings containing a given string as a substring is recognisable.

Chapter 4

ε-automata

Non-deterministic and deterministic automata have something in common: both types of machines can only change state in response to reading an input symbol. In the case of non-deterministic automata, a state and an input symbol lead to a set of possible states. The class of ε-automata, introduced in this chapter, can change state spontaneously without any input symbol being read. Although this sounds like a powerful feature, we shall show that every non-deterministic automaton with ε-transitions can be converted into a non-deterministic automaton without ε-transitions that recognises the same language. Armed with ε-automata, we can construct automata to recognise all kinds of languages with great ease.

4.1 Automata with ε-transitions

In both deterministic and non-deterministic automata, transitions may only be labelled with elements of the input alphabet. No edge may be labelled with the empty string ε. We shall now waive this restriction. A *non-deterministic automaton with ε-transitions* or, more simply, an *ε-automaton*, is a 5-tuple,

$$\mathbf{A} = (S, A, I, \delta_\varepsilon, T),$$

where all the symbols have the same meanings as in the non-deterministic case except that

$$\delta_\varepsilon \colon S \times (A \cup \{\varepsilon\}) \to \mathsf{P}(S).$$

As before, we shall write $\delta_\varepsilon(s, a) = s \cdot a$. The only difference between such automata and non-deterministic automata is that we allow transitions:

$$\begin{tikzcd}
s \arrow[r, "\varepsilon"] & t
\end{tikzcd}$$

Such transitions are called *ε-transitions.*

In order to define what we mean by the language accepted by such a machine, we have to define an appropriate 'extended transition function.' This is slightly more involved than before, so I shall begin with an informal description. A path in an ε-automaton is a sequence of states each labelled by an element of the set $A \cup \{\varepsilon\}$. The string corresponding to this path is the *concatenation* of these labels in order. We say that a string x is accepted by an ε-automaton if there is a path from an initial state to a terminal state the concatenation of whose labels is x. I now have to put this idea on a sound footing.

Let A be an alphabet. If $a \in A$, then for all $m, n \in \mathbb{N}$ we have that $a = \varepsilon^m a \varepsilon^n$. However, $\varepsilon^m a \varepsilon^n$ is also a *string* consisting of m ε's followed by one a followed by a further n ε's. We call this string an *ε-extension* of the symbol a. The *value* of the ε-extension $\varepsilon^m a \varepsilon^n$ is a. More generally, we can define an ε-extension of a string $x \in A^*$ to be the product of ε-extensions of each symbol in x. The value of any such ε-extension is just x. For example, the string aba has ε-extensions of the form $\varepsilon^m a \varepsilon^n b \varepsilon^p a \varepsilon^q$, where $m, n, p, q \in \mathbb{N}$. Let $\mathbf{A}$ be a non-deterministic automaton with ε-transitions. We say that x is *accepted by* $\mathbf{A}$ if *some* ε-extension of x labels a path in $\mathbf{A}$ starting at some initial state and ending at some terminal state. As usual we write $L(\mathbf{A})$ to mean the set of all strings accepted by $\mathbf{A}$. It is now time for a concrete example.

Example 4.1.1 Consider the diagram below:

$$\begin{tikzcd}
{} \arrow[r] & p \arrow[r, shift left, "\varepsilon"] \arrow[d, "b"'] & q \arrow[l, shift left, "b"] \arrow[r, "a"] & r \arrow[dl, shift left, "a"] \\
& s \arrow[r, "\varepsilon"'] & t \arrow[ur, shift left, "\varepsilon"] \arrow[r, "b"'] & u \arrow[u, "\varepsilon"']
\end{tikzcd}$$

This is clearly a non-deterministic automaton with ε-transitions. We find some examples of strings accepted by this machine. First of all, the letter a is accepted. At first sight, this looks wrong, because there are no transitions from p to r labelled by a. However, this is not our definition of how a string is accepted. We have to check *all possible ε-extensions of a*. In this case, we immediately see that εa labels a path from p to r, and so a *is* accepted. Notice, by the way, that it is the *value* of the ε -extension that is accepted; so, if you said εa was accepted, I would have to say that you were wrong. The letter b is accepted, because $b\varepsilon\varepsilon$ labels a path from p to r. The string bb is accepted, because $b\varepsilon b\varepsilon$ labels a path from p to r. □

Now that we understand how ε-automata are supposed to behave, we can formally define the extended transition function δ_ε^*. To do this, we shall use the following definition. Let $\mathbf{A}$ be a non-deterministic automaton with ε-transitions, and let s be an arbitrary state of $\mathbf{A}$. The *ε-closure of s*, denoted by $E(s)$, consists of s itself together with all states in $\mathbf{A}$, which can be reached by following paths labelled *only* by ε's. If Q is a set of states, then we define the *ε-closure of Q* by

$$E(Q) = \sum_{q \in Q} E(q),$$

the union of the ε-closures of each of the states in Q. Observe that $E(\emptyset) = \emptyset$. Referring to Example 4.1.1, the reader should check that $E(p) = \{p, q\}$, $E(q) = \{q\}$, $E(r) = \{r\}$, $E(s) = \{s, t, r\}$, $E(t) = \{t, r\}$, and $E(u) = \{u, r\}$. The only point that needs to be emphasised is that the ε-closure of a state must contain that state, and so it can never be empty.

We are almost ready to define the extended transition function. We need one piece of notation.

Notation If Q is a set of states in an ε-automaton and $a \in A$ then we write $Q \cdot a$ to mean $\sum_{q \in Q} q \cdot a$; that is, a state s belongs to the set $Q \cdot a$ precisely when there is a state $q \in Q$ and a transition in $\mathbf{A}$ from q to s labelled by a.

The *extended transition function* of an ε-automaton δ_ε^* is the unique function from $S \times A^*$ to $\mathsf{P}(S)$ satisfying the following three conditions where $a \in A$, $x \in A^*$ and $s \in S$:

(ETF1) $\delta_\varepsilon^*(s, \varepsilon) = E(s)$.

(ETF2) $\delta_\varepsilon^*(s, a) = E(E(s) \cdot a)$.

(ETF3) $\delta_\varepsilon^*(s, ax) = \sum_{q \in E(E(s) \cdot a)} \delta_\varepsilon^*(q, x)$.

Once again, it can be shown that this defines a unique function. This definition agrees perfectly with our definition of the ε-extension of a string. To see why, observe that if $a \in A$, then $E(E(s) \cdot a)$ is the set of states that can be reached starting at s and following all paths labelled $\varepsilon^m a \varepsilon^n$. More generally, $\delta_\varepsilon^*(s, x)$, where $x \in A^*$, consists of all states that can be reached starting at s and following all paths labelled by ε-extensions of x. We conclude that the appropriate definition of the language accepted by an ε-automaton is

$$L(\mathbf{A}) = \{x \in A^* \colon \delta_\varepsilon^*(s, x) \cap T \neq \emptyset \text{ for some } s \in I\}.$$

Our goal now is to show that a language recognised by an ε-automaton can be recognised by an ordinary non-deterministic automaton. To do this, we shall use the following construction. Let $\mathbf{A} = (S, A, I, \delta_\varepsilon, T)$ be a non-deterministic automaton with ε-transitions. Define a non-deterministic automaton,

$$\mathbf{A}^s = (S \cup \{\diamondsuit\}, A, I \cup \{\diamondsuit\}, \Delta, T^s),$$

as follows:

- $\diamondsuit$ is a new state.
- $$T^s = \begin{cases} T \cup \{\diamondsuit\} & \text{if } \varepsilon \in L(\mathbf{A}) \\ T & \text{otherwise.} \end{cases}$$
- The function,
  $$\Delta \colon (S \cup \{\diamondsuit\}) \times A \to \mathsf{P}(S \cup \{\diamondsuit\}),$$
  is defined as follows: $\Delta(\diamondsuit, a) = \emptyset$ for all $a \in A$, and $\Delta(s, a) = E(E(s) \cdot a)$ for all $s \in S$ and $a \in A$.

It is clear that $\mathbf{A}^s$ is a well-defined, non-deterministic automaton. Observe that the role of the state $\diamondsuit$ is solely to accept ε if $\varepsilon \in L(\mathbf{A})$. If $\varepsilon \notin L(\mathbf{A})$, then you can omit $\diamondsuit$ from the construction of $\mathbf{A}^s$.

Theorem 4.1.2 *Let $\mathbf{A} = (S, A, I, \delta_\varepsilon, T)$ be a non-deterministic automaton with ε-transitions. Then $L(\mathbf{A}^s) = L(\mathbf{A})$.*

Proof The main plank of the proof is the following equation, which we shall prove below: for all $s \in S$ and $x \in A^+$ we have that

$$\Delta^*(s, x) = \delta_\varepsilon^*(s, x). \tag{4.1}$$

Observe that this equation holds for *non-empty* strings. We prove (4.1) by induction on the length of x.

For the base step, let $a \in A$. Then

$$\Delta^*(s,a) = \Delta(s,a) = \delta_\varepsilon^*(s,a),$$

simply following the definitions.

For the induction step, assume that the equality holds for all $x \in A^+$ where $|\,x\,| = n$. Let $y = ax$ where $a \in A$ and $|\,x\,| = n$. Then

$$\Delta^*(s,y) = \Delta^*(s,ax) = \sum_{q \in \Delta(s,a)} \Delta^*(q,x),$$

by the definition of Δ^*. By the base step and the induction step,

$$\sum_{q \in \Delta(s,a)} \Delta^*(q,x) = \sum_{q \in \delta_\varepsilon^*(s,a)} \delta_\varepsilon^*(q,x),$$

but by definition,

$$\sum_{q \in \delta_\varepsilon^*(s,a)} \delta_\varepsilon^*(q,x) = \delta_\varepsilon^*(s,ax) = \delta_\varepsilon^*(s,y),$$

and we have proved the equality.

Now we can prove that $L(\mathbf{A}^s) = L(\mathbf{A})$. Observe first that

$$\varepsilon \in L(\mathbf{A}) \Leftrightarrow \diamondsuit \in T^s \Leftrightarrow \varepsilon \in L(\mathbf{A}^s).$$

With this case out of the way, let $x \in A^+$. Then by definition $x \in L(\mathbf{A}^s)$ means that there is some $s \in I \cup \{\diamondsuit\}$ such that $\Delta^*(s,x) \cap T^s \neq \emptyset$. But since x is not empty, the state $\diamondsuit$ can play no role and so we can write that for some $s \in I$, we have $\Delta^*(s,x) \cap T \neq \emptyset$. By equation (4.1), $\Delta^*(s,x) = \delta_\varepsilon^*(s,x)$. Thus $x \in L(\mathbf{A}^s)$ if and only if $\delta_\varepsilon^*(s,x) \cap T \neq \emptyset$ for some $s \in I$. This of course says precisely that $x \in L(\mathbf{A})$ as required. $\square$

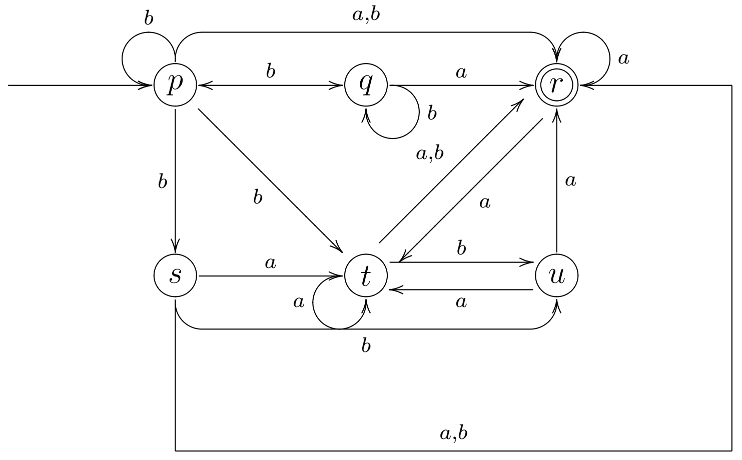
Remark The meaning of the 's' in $\mathbf{A}^s$ is that of 'sans' since $\mathbf{A}^s$ is 'sans epsilons.'

The construction of the machine $\mathbf{A}^s$ is quite involved. It is best to set out the calculations in tabular form as suggested by the following example.

Example 4.1.3 We calculate **A** for the ε-automaton of Example 4.1.1.

state $\star$	$E(\star)$	$E(\star)\cdot a$	$E(\star)\cdot b$	$E(E(\star)\cdot a)$	$E(E(\star)\cdot b)$
p	$\{p,q\}$	$\{r\}$	$\{s,p\}$	$\{r\}$	$\{s,t,r,p,q\}$
q	$\{q\}$	$\{r\}$	$\{p\}$	$\{r\}$	$\{p,q\}$
r	$\{r\}$	$\{t\}$	$\emptyset$	$\{t,r\}$	$\emptyset$
s	$\{s,t,r\}$	$\{t\}$	$\{u\}$	$\{t,r\}$	$\{u,r\}$
t	$\{t,r\}$	$\{t\}$	$\{u\}$	$\{t,r\}$	$\{u,r\}$
u	$\{u,r\}$	$\{t\}$	$\emptyset$	$\{t,r\}$	$\emptyset$

The last two columns give us the information required to construct $\mathbf{A}^s$ below:

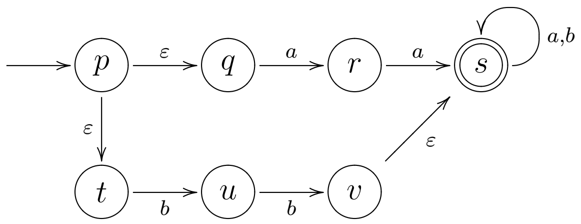


In this case, the state labelled $\diamond$ is omitted because the original automaton does not accept the empty string. $\square$

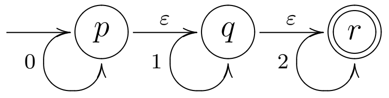
Exercises 4.1

1. For each of the ε-automata $\mathbf{A}$ below construct $\mathbf{A}^s$ and $\mathbf{A}^{sda} = ((\mathbf{A}^s)^d)^a$. In each case, describe $L(\mathbf{A})$.

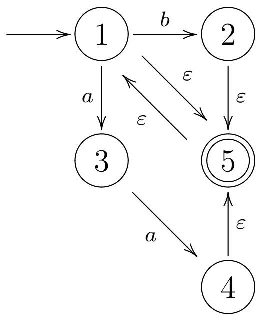
(i)



(ii)



(iii)



4.2 Applications of ε-automata

If L and M are both recognisable, then ε-automata provide a simple way of proving that $L + M$, LM and L^* are all recognisable.

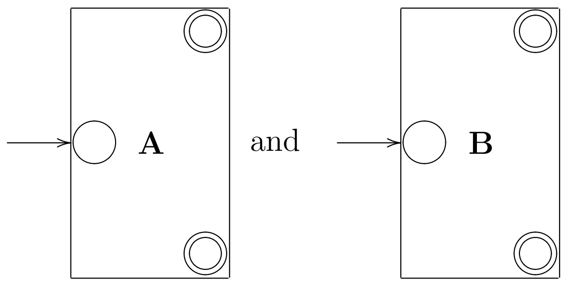
Theorem 4.2.1 *Let A be an alphabet and L and M be languages over A.*

(i) *If L and M are recognisable then $L + M$ is recognisable.*

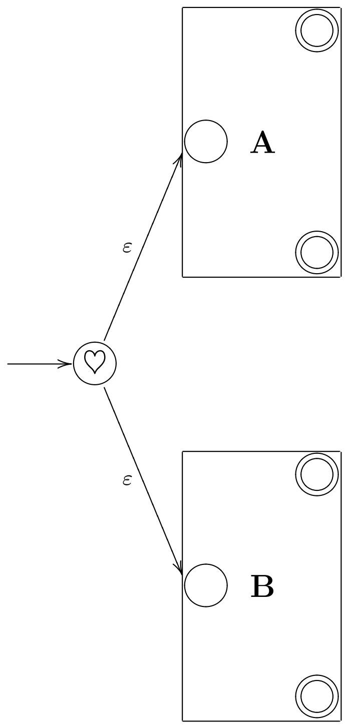
(ii) *If L and M are recognisable then LM is recognisable.*

(iii) *If L is recognisable then L^* is recognisable.*

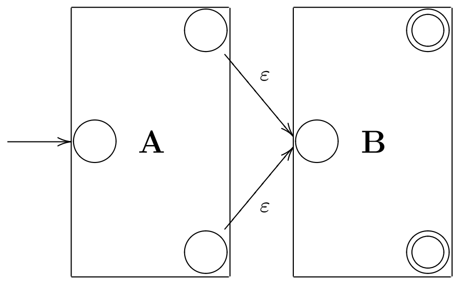
Proof (i) By assumption, we are given two automata **A** and **B** such that $L(\mathbf{A}) = L$ and $L(\mathbf{B}) = M$. Construct the following ε-automaton: introduce a new state, which we label $\heartsuit$, to be the new initial state and draw ε-transitions to the initial state of **A** and the initial state of **B**; the initial states of **A** and **B** are now converted into ordinary states. Call the resulting ε-automaton **C**. It is clear that this machine recognises the language $L + M$. We now apply Theorem 4.1.2 to obtain a non-deterministic automaton recognising $L + M$. Thus $L + M$ is recognisable. If we picture **A** and **B** schematically as follows:



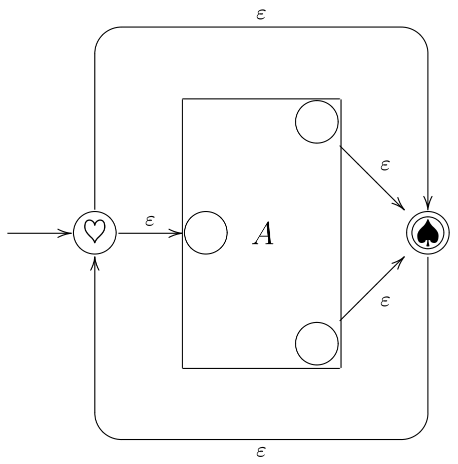
Then the machine **C** has the following form:



(ii) By assumption, we are given two automata **A** and **B** such that $L(\mathbf{A}) = L$ and $L(\mathbf{B}) = M$. Construct the following ε-automaton: from each terminal state of **A** draw an ε-transition to the initial state of **B**. Make each of the terminal states of **A** ordinary states and make the initial state of **B** an ordinary state. Call the resulting automaton **C**. It is easy to see that this ε-automaton recognises LM. We now apply Theorem 4.1.2 to obtain a non-deterministic automaton recognising LM. Thus LM is recognisable. If we picture **A** and **B** schematically as above then the machine **C** has the following form:



(iii) Let **A** be a deterministic automaton such that $L(\mathbf{A}) = L$. Construct an ε-automaton **D** as follows. It has two more states than **A**, which we shall label ♡ and ♠: the former will be initial and the latter terminal. Connect the state ♡ by an ε-transition to the initial state of **A** and then make the initial state of **A** an ordinary state. Connect all terminal states of **A** to the state labelled ♠ and then make all terminal states of **A** ordinary states. Connect the state ♡ by an ε-transition to the state ♠ and vice versa. If we picture **A** schematically as above, then **D** can be pictured schematically as follows:



It is easy to check that $L(\mathbf{B}) = L^*$: first, by construction, ε is recognised. Second, the bottom ε-transition enables us to re-enter the machine $\mathbf{A}$ embedded in the diagram. The result now follows by Theorem 4.1.2 again. □

Exercises 4.2

1. Construct ε-automata to recognise each of the following languages.

(i) $(a^2)^*(b^3)^*$.

(ii) $(a(ab)^*b)^*$.

(iii) $(a^2b^* + b^2a^*)(ab + ba)$.

4.3 Summary of Chapter 4

- *ε-automata*: These are defined just as non-deterministic automata except that we also allow transitions to be labelled by ε. A string x over the input alphabet is accepted by such a machine $\mathbf{A}$ if there is at least one path in $\mathbf{A}$ starting at an initial state and finishing at a terminal state such that when the labels on this path are concatenated the string x is obtained.

- $\mathbf{A}^s$: There is an algorithm that converts an ε-automaton $\mathbf{A}$ into a non-deterministic automaton $\mathbf{A}^s$ recognising the same language. The 's' stands for 'sans' meaning 'without (epsilons).'

- *Applications*: Using ε-automata, simple proofs can be given of the recognisability of LM from the recognisability of L and M, and the recognisability of L^* from the recognisability of L.

Chapter 5

Kleene's Theorem

Chapters 2 to 4 have presented us with an array of languages that we can show to be recognisable. At the same time, we have seen that there are languages that are not recognisable. It is clearly time to find a characterisation of recognisable languages. This is exactly what Kleene's theorem does. The characterisation is in terms of regular expressions. Such expressions form a notation for describing languages in terms of finite languages, union, product, and Kleene star; it was informally introduced in Section 1.3. Kleene's theorem states that a language is recognisable precisely when it can be described by a regular expression.

5.1 Regular languages

This is now a good opportunity to reflect on which languages we can now prove are recognisable. I want to pick out four main results:

- Finite languages are recognisable; this was proved in Proposition 2.3.4.
- The union of two recognisable languages is recognisable; this was proved in Proposition 2.6.6.
- The product of two recognisable languages is recognisable; this was proved in Proposition 4.2.1(ii).
- The Kleene star of a recognisable language is recognisable; this was proved in Proposition 4.2.1(iii).

We now analyse these results a little more deeply. A finite language that is neither empty nor consists of just the empty string is a finite union of strings, and each language consisting of a finite string is a finite product of languages each of which consist of a single letter. Call a language over an alphabet *basic* if it is either empty, consists of the empty string alone, or consists of a single symbol from the alphabet. Then what we have proved is the following: a language that can be constructed from the basic languages by using only the operations $+$, $\cdot$ and $*$ a finite number of times must be recognisable. The following two definitions give a precise way of describing such languages.

Let $A = \{a_1, \ldots, a_n\}$ be an alphabet. A *regular expression over A* (the term *rational expression* is also used) is a sequence of symbols formed by repeated application of the following rules:

(R1) $\emptyset$ is a regular expression.

(R2) ε is a regular expression.

(R3) $a_1, \ldots, a_n$ are each regular expressions.

(R4) If s and t are regular expressions then so is $(s + t)$.

(R5) If s and t are regular expressions then so is $(s \cdot t)$.

(R6) If s is a regular expression then so is (s^*).

(R7) Every regular expression arises by a finite number of applications of the rules (R1) to (R6).

We call $+$, $\cdot$, and $*$ the *regular operators*. As usual, we will generally write st rather than $s \cdot t$. It is easy to determine whether an expression is regular or not.

Example 5.1.1 We claim that $((0 \cdot (1^*)) + 0)$ is a regular expression over the alphabet $\{0, 1\}$. To prove that it is, we simply have to show that it can be constructed according to the rules above:

(1) 1 is regular by (R3).

(2) (1^*) is regular by (R6).

(3) 0 is regular by (R3).

(4) $(0 \cdot (1^*))$ is regular by (R5) applied to (2) and (3) above.

(5) $((0 \cdot (1^*)) + 0)$ is regular by (R4) applied to (4) and (3) above.

$\square$

Each regular expression s describes a language, denoted by $L(s)$. This language is calculated by means of the following rules, which agree with the conventions we introduced in Section 1.3. Simply put, they tell us how to 'insert the curly brackets.'

(D1) $L(\emptyset) = \emptyset$.

(D2) $L(\varepsilon) = \{\varepsilon\}$.

(D3) $L(a_i) = \{a_i\}$.

(D4) $L(s + t) = L(s) + L(t)$.

(D5) $L(s \cdot t) = L(s) \cdot L(t)$.

(D6) $L(s^*) = L(s)^*$.

Now that we know how regular expressions are to be interpreted, we can introduce some conventions that will enable us to remove many of the brackets, thus making regular expressions much easier to read and interpret. The way we do this takes its cue from ordinary algebra. For example, consider the algebraic expression $a+bc^{-1}$. This can only mean $a+(b(c^{-1}))$, but $a+bc^{-1}$ is much easier to understand than $a + (b(c^{-1}))$. If we say that $*$, $\cdot$, and $+$ behave, respectively, like $^{-1}$, $\times$, and $+$ in ordinary algebra, then we can, just as in ordinary algebra, dispense with many of the brackets that the definition of a regular expression would otherwise require us to use. Using this convention, the regular expression $((0 \cdot (1^*)) + 0)$ would usually be written as $01^* + 0$. Our convention tells us that 01^* means $0(1^*)$ rather than $(01)^*$, and that $01^* + 0$ means $(01^*) + 0$ rather than $0(1^* + 0)$.

Example 5.1.2 We calculate $L(01^* + 0)$.

(1) $L(01^* + 0) = L(01^*) + L(0)$ by (D4).

(2) $L(01^*) + L(0) = L(01^*) + \{0\}$ by (D3).

(3) $L(01^*) + \{0\} = L(0) \cdot L(1^*) + \{0\}$ by (D5).

(4) $L(0) \cdot L(1^*) + \{0\} = \{0\} \cdot L(1^*) + \{0\}$ by (D3).

(5) $\{0\} \cdot L(1^*) + \{0\} = \{0\} \cdot L(1)^* + \{0\}$ by (D6).

(6) $\{0\} \cdot L(1)^* + \{0\} = \{0\} \cdot \{1\}^* + \{0\}$ by (D3).

□

Two regular expressions s and t are *equal*, written $s = t$, if and only if $L(s) = L(t)$. Two regular expressions can look quite different yet describe the same language and so be equal.

Example 5.1.3 Let $s = (0+1)^*$ and $t = (1 + 00^*1)^*0^*$. We shall show that these two regular expressions describe the same language. Consequently,

$$(0+1)^* = (1 + 00^*1)^*0^*.$$

We now prove this assertion. Because $(0+1)^*$ describes the language of all possible strings of 0's and 1's it is clear that $L(t) \subseteq L(s)$. We need to prove the reverse inclusion. Let $x \in (0+1)^*$, and let u be the longest prefix of x belonging to 1^*. Put $x = ux'$. Either $x' \in 0^*$, in which case $x \in L(t)$, or x' contains at least one 1. In the latter case, x' begins with a 0 and contains at least one 1. Let v be the longest prefix of x' from 0^+1. We can therefore write $x = uvx''$ where $u \in 1^*$, $v \in 0^+1$ and $|\,x''\,| < |\,x\,|$. We now replace x by x'' and repeat the above process. It is now clear that $x \in L(t)$. □

A language L is said to be *regular* (the term *rational* is also used) if there is a regular expression s such that $L = L(s)$.

Examples 5.1.4 Here are a few examples of regular expressions and the languages they describe over the alphabet $A = \{a, b\}$.

(1) Let $L = \{x \in (a+b)^* \colon |\,x\,| \text{ is even}\}$. A string of even length is either just ε on its own or can be written as the concatenation of strings each of length 2. Thus this language is described by the regular expresssion $((a+b)^2)^*$.

(2) Let $L = \{x \in (a+b)^* \colon |\,x\,| \equiv 1 \pmod 4\}$. A string belongs to this language if its length is one more than a multiple of 4. A string of length a multiple of 4 can be described by the regular expression $((a+b)^4)^*$. Thus a regular expression for L is $((a+b)^4)^*(a+b)$.

(3) Let $L = \{x \in (a+b)^*\colon |\,x\,| < 3\}$. A string belongs to this language if its length is $0, 1$, or 2. A suitable regular expression is therefore $\varepsilon + (a+b) + (a+b)^2$. The language L', the complement of L, consists of all strings whose length is at least 3. This language is described by the regular expression $(a+b)^3(a+b)^*$.

□

We have seen that two regular expressions s and t may look different but describe the same language $L(s) = L(t)$ and so be equal as regular expressions. The collection of all languages $\mathsf{P}(A^*)$ has a number of properties that are useful in showing that two regular expressions are equal. The simplest ones are described in the proposition below. The proofs are left as exercises.

Proposition 5.1.5 *Let A be an alphabet, and let $L, M, N \in \mathsf{P}(A^*)$. Then the following properties hold:*

(i) $L + (M + N) = (L + M) + N$.

(ii) $\emptyset + L = L = L + \emptyset$.

(iii) $L + L = L$.

(iv) $L \cdot (M \cdot N) = (L \cdot M) \cdot N$.

(v) $\varepsilon \cdot L = L = L \cdot \varepsilon$.

(vi) $\emptyset \cdot L = \emptyset = L \cdot \emptyset$.

(vii) $L \cdot (M + N) = L \cdot M + L \cdot N$, *and* $(M + N) \cdot L = M \cdot L + N \cdot L$. □

Result (i) above is called the *associativity law* for unions of languages, whereas result (iv) is the associativity law for products of languages. Result (vii) contains the two *distributity laws (left* and *right* respectively*)* for product over union.

Because equality of regular expressions $s = t$ is defined in terms of the equality of the corresponding languages $L(s) = L(t)$ it follows that the seven properties above also hold for regular expressions. A few examples are given below.

Examples 5.1.6 Let r, s and t be regular expressions. Then

(1) $r + (s + t) = (r + s) + t$.

(2) $(rs)t = r(st)$.

(3) $r(s + t) = rs + rt$.

□

The relationship between the Kleene star and the other two regular operators is much more complex. Here are two examples.

Examples 5.1.7 Let $A = \{a, b\}$.

(1) $(ab)^* = \varepsilon + a(ba)^*b$. The left-hand side is

$$\varepsilon + (ab) + (ab)^2 + (ab)^3 + \dots.$$

However, for $n \geq 1$, the string $(ab)^n$ is equal to $a(ba)^{n-1}b$. Thus the left-hand side is equal to the right-hand side.

(2) $(a+b)^* = (a^*b)^*a^*$. To prove this we apply the usual method for showing that two sets X and Y are equal: we show that $X \subseteq Y$ and $Y \subseteq X$. It is clear that the language on the right is a subset of the language on the left. We therefore need only explicitly prove that the language on the left is a subset of the language on the right. A typical term of $(a + b)^*$ consists of a finite product of a's and b's. Either this product consists entirely of a's, in which case it is clearly a subset of the right-hand side, or it also contains at least one b: in which case, we can split the product into sequences of a's followed by a b, and possibly a sequence of a's at the end. This is also a subset of the right-hand side. For example,

$$aaabbabaaabaaa$$

can be written as

$$(aaab)(a^0b)(ab)(aaab)aaa,$$

which is clearly a subset of $(a^*b)^*a^*$.

□

Exercises 5.1

1. Find regular expressions for each of the languages over $A = \{a, b\}$.

   (i) All strings in which a always appears in multiples of 3.

   (ii) All strings that contain exactly 3 a's.

   (iii) All strings that contain exactly 2 a's or exactly 3 a's.

   (iv) All strings that do not contain aaa.

   (v) All strings in which the total number of a's is divisible by 3.

   (vi) All strings that end in a double letter.

   (vii) All strings that have exactly one double letter.

2. Let r and s be regular expressions. Prove that each of the following equalities holds between the given pair of regular expressions.

   (i) $r^* = (rr)^* + r(rr)^*$.

   (ii) $(r + s)^* = (r^*s^*)^*$.

   (iii) $(rs)^*r = r(sr)^*$.

3. Prove Proposition 5.1.5.

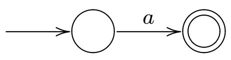
5.2 An algorithmic proof of Kleene's theorem

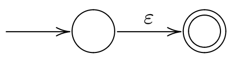
In this section, we shall describe two algorithms that together provide an algorithmic proof of Kleene's theorem: our first algorithm will show explicitly how to construct an ε-automaton from a regular expression, and our second will show explicitly how to construct a regular expression from an automaton.

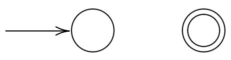
In the proof below we shall use a class of ε-automata. A *normalised ε-automaton* is just an ε-automaton having exactly one initial state and one terminal state, and the property that there are no transitions into the initial state or out of the terminal state.

Theorem 5.2.1 (Regular expression to ε-automaton) *Let r be a regular expression over the alphabet A. Let m be the sum of the following two numbers: the number of symbols from A occurring in r, counting repeats, and the number of regular operators occurring in r, counting repeats. Then there is an ε-automaton $\mathbf{A}$ having at most $2m$ states such that $L(\mathbf{A}) = L$.*

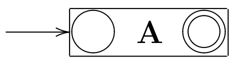
Proof We shall prove that each regular language is recognised by some normalised ε-automaton satisfying the conditions of the theorem. Base step: prove that if $L = L(r)$ where r is a regular expression without regular operators, then L can be recognised by a normalised ε-automaton with at most 2 states. However, in this case L is either $\{a\}$ where $a \in A$, $\emptyset$, or $\{\varepsilon\}$. The normalised ε-automata, which recognise each of these languages, are



and


and


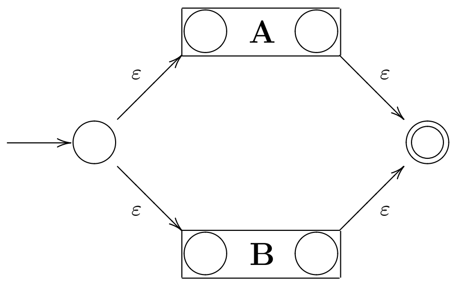
Induction hypothesis: assume that if r is a regular expression, using at most $n-1$ regular operators and containing p occurrences of letters from the underlying alphabet, then $L(r)$ can be recognised by a normalised ε-automaton using at most $2(n-1)+2p$ states. Now let r be a regular expression having n regular operators and q occurrences of letters from the underlying alphabet. We shall prove that $L(r)$ can be recognised by a normalised ε-automaton containing at most $2n+2q$ states. From the definition of a regular expression, r must have one of the following three forms: (1) $r = s + t$, (2) $r = s \cdot t$ or (3) $r = s^*$. Clearly, s and t each use at most $n-1$ regular operators; let n_s and n_t be the number of regular operators occurring in s and t, respectively, and let q_s and q_t be the number of occurrences of letters from the underlying alphabet in s and t, respectively. Then $n_s + n_t = n - 1$ and $q_s + q_t = q$. So by the induction hypothesis $L(s)$ and $L(t)$ are recognised by normalised ε-automata **A** and **B**, respectively, which have at most $2(n_s + q_s)$ and $2(n_t + q_t)$ states apiece. We can picture these as follows:



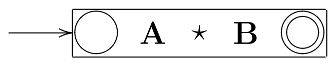
and

We now show how **A** and **B** can be used to construct ε-automata to recognise each of the languages described by the regular expressions (1), (2), and (3), respectively; our constructions are just mild modifications of the ones used in Theorem 4.2.1.

(1) A normalised ε-automaton recognising $L(s + r)$ is

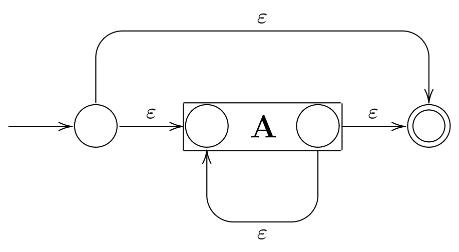


(2) A normalised ε-automaton recognising $L(s \cdot t)$ is



This automaton is obtained by merging the terminal state of **A** with the initial state of **B** and making the resulting state, marked with a $\star$, an ordinary state.

(3) A normalised ε-automaton recognising $L(s^*)$ is

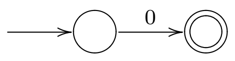


In all three cases, the number of states in the resulting machine is at most

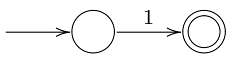
$$2(n_s + q_s) + 2(n_t + q_t) + 2 = 2(n + q),$$

which is the answer required. $\square$

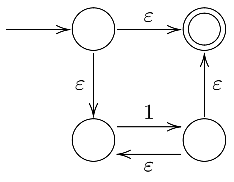
Example 5.2.2 Here is an example of Theorem 5.2.1 in action. Consider the regular expression $01^* + 0$. To construct an ε-automaton recognising the language described by this regular expression, we begin with the two automata



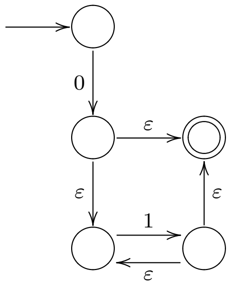
and



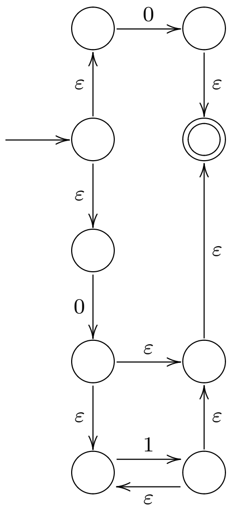
We convert the second machine into one recognising 1^*:



We then combine this machine with our first to obtain a machine recognising 01^*:



We now combine this machine with the one for 0 to obtain the following machine respresenting $01^* + 0$:



□

Remark Using the algorithm of Theorem 5.2.1 will lead to large machines for even quite small regular expressions. A better option is not to break up the regular expression into its component letters but instead into regular expressions which can easily be converted into automata. For example, suppose we want to construct an ε-machine recognising $a^*(ba^*)^*$. We can think of this as the product of a^* and $(ba^*)^*$. The latter is the Kleene star of ba^*. It is easy to construct machines recognising a^* and ba^* respectively. The machine for ba^* can easily be converted into one recognising $(ba^*)^*$, and this latter machine can be combined with the one for a^* to yield a machine recognising our original regular expression. The decomposition of the regular expression we used can be regarded as a tree:

$$\begin{tikzcd}
 & a^*(ba^*)^* \arrow[dl, dash] \arrow[dr, dash] & \\
a^* & & (ba^*)^* \arrow[d, dash] \\
 & & ba^*
\end{tikzcd}$$

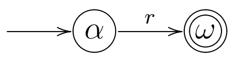
and this tree can be used as a guide in contructing the associated automaton.

We shall now show how to construct a regular expression from an automaton. To do so, it is convenient to introduce a yet more general class of automata than even the ε-automata.

A *generalised automaton* over the input alphabet A is the same as an ε-automaton except that we allow the transitions to be labelled by arbitrary regular expressions over A. The language $L(\mathbf{A})$ recognised by a generalised automaton $\mathbf{A}$ is defined as follows. Let $x \in A^*$. Then $x \in L(\mathbf{A})$ if there is a path in $\mathbf{A}$, which begins at one of the initial states, ends at one of the terminal states, and whose labels are, in order, the regular expressions $r_1, \ldots, r_n$, such that x can be factorised $x = x_1 \ldots x_n$ in such a way that each $x_i \in L(r_i)$. The definition of $L(\mathbf{A})$ generalises the definition of the language recognised by an ε-automaton. To use generalised automata to find the regular expression describing the language recognised by an automaton, we shall need the following. A *normalised (generalised) automaton* is a generalised automaton with exactly one initial state, which I shall always label α, and exactly one terminal state, which I shall always label ω; in addition, there are no transitions into α nor any transitions out of ω. A normalised generalised automaton is therefore a substantial generalisation of a normalised ε-automaton used in the proof of Theorem 5.2.1. Every generalised automaton can easily be normalised in such a way that the language is not changed: adjoin a new initial state with transitions labelled ε pointing at the old initial states, and adjoin a new terminal state with transitions from each of the old terminal states labelled ε pointing at the new terminal state.

Terminology For the remainder of this section, 'normalised automaton' will always mean a 'normalised generalised automaton'.

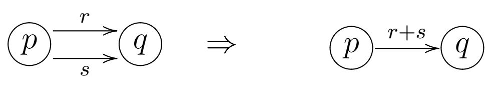
The simplest kinds of normalised automata are those with exactly one initial state, one terminal state, and at most one transition:



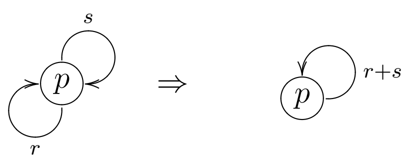
We call such a normalised automaton *trivial*. If a trivial automaton has a transition, then the label of that transition will be a regular expression, and this regular expression obviously describes the language accepted by the automaton; if there is no transition then the language is $\emptyset$.

We shall describe an algorithm that converts a normalised automaton into a trivial normalised automaton recognising the same language. The algorithm depends on three operations, which may be carried out on a normalised automaton that we now describe.

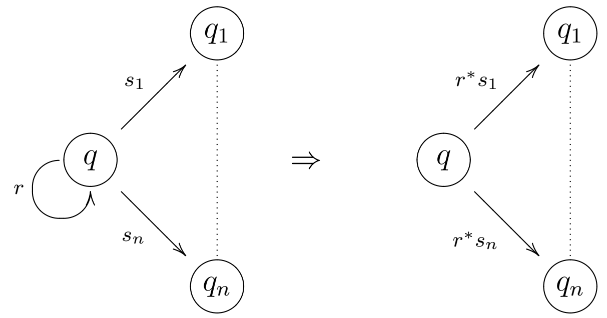
(T) transition elimination Given any two states p and q, where $p = q$ is not excluded, all the transitions from p to q can be replaced by a single transition by applying the following rule:



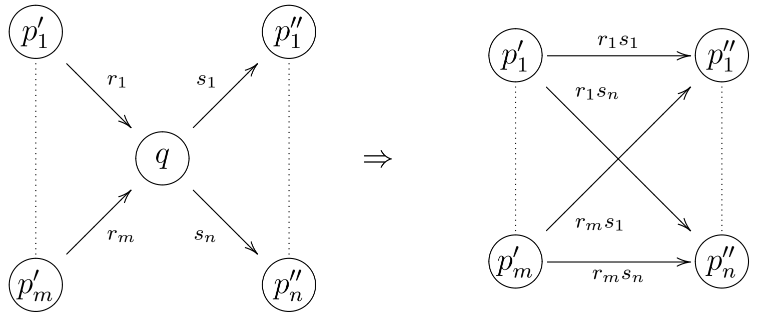
In the case where $p = q$, this rule takes the following form:



(L) loop elimination Let q be a state that is neither α nor ω, and suppose this state has a single loop labelled r. If there are no transitions entering this state or no transitions leaving this state, then q can be erased together with any transitions from it or any transitions to it. We may therefore restrict our attention to the case where q has at least one transition entering it and at least one transition leaving it. In this case, the loop at q can be erased, and for each transition leaving q labelled by s we change the label to r^*s. This operation is summarised in the diagram below:



(S) state elimination Let q be a state that is neither α nor ω and that has no loop. If there are no transitions entering this state or no transitions leaving this state, then q can be erased together with any transitions from it or any transitions to it. We may therefore restrict our attention to the case where q has at least one transition entering it and at least one transition leaving it. In this case, we do the following: for each transition $p' \xrightarrow{r} q$ and for each transition $q \xrightarrow{s} p''$, both of which I shall call 'old,' we construct a 'new' transition $p' \xrightarrow{rs} p''$. At the end of this process the state q and all the old transitions are erased. This operation is summarised in the diagram below:



Lemma 5.2.3 *Let* **A** *be a normalised automaton, and let* **B** *be the normalised automaton that results when one of the rules* (T), (L) *or* (S) *is applied. Then* $L(\mathbf{B}) = L(\mathbf{A})$.

Proof We simply check each case in turn. This is left as an exercise. □

These operations are the basis of the following algorithm.

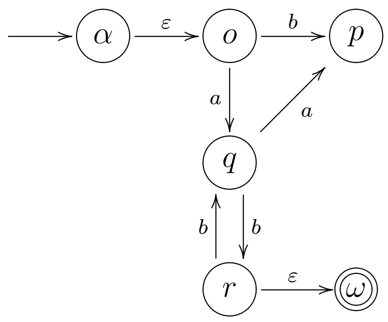
Algorithm 5.2.4 (Automaton to regular expression) The input to this algorithm is a normalised automaton $\mathbf{A}$. The output is a regular expression r such that $L(r) = L(\mathbf{A})$.

Repeat the following procedure until there are only two states and at most one transition between them, at which point the algorithm terminates.

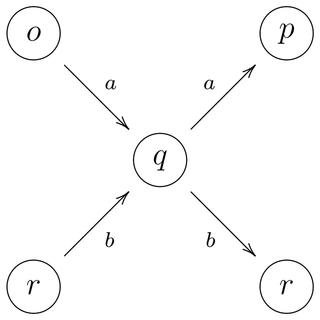
Procedure: repeatedly apply rule (T) if necessary until the resulting automaton has the property that between each pair of states there is at most one transition; now repeatedly apply rule (L) if necessary to eliminate all loops; finally, apply rule (S) to eliminate a state.

When the algorithm has terminated, a regular expression describing the language recognised by the original machine is given by the label of the unique transition, if there is a transition, otherwise the language is the empty set. □

Example 5.2.5 Consider the automaton:



The rules (T) and (L) are superfluous here, so we shall go straight to rule (S) applied to the state q. The pattern of incoming and outgoing transitions for this state is



If we now apply rule (S), we obtain the following pattern of transitions:

$$\begin{tikzcd}
o \arrow[r, "a^2"] \arrow[dr, "ab"'] & p \\
r \arrow[ur, "ba"] \arrow[r, "b^2"'] & r
\end{tikzcd}$$

The resulting generalised automaton is therefore

$$\begin{tikzcd}
{} \arrow[r] & \alpha \arrow[r, "\varepsilon"] & o \arrow[r, shift left, "a^2"] \arrow[r, shift right, "b"'] \arrow[d, "ab"'] & p \\
& & r \arrow[ur, "ba"'] \arrow[r, "\varepsilon"] \arrow[loop left, "b^2"] & \omega
\end{tikzcd}$$

If we apply the rules (T) and (L) to this generalised automaton we get

$$\begin{tikzcd}
{} \arrow[r] & \alpha \arrow[r, "\varepsilon"] & o \arrow[r, "a^2+b"] \arrow[d, "ab"'] & p \\
& & r \arrow[ur, "(b^2)^*ba"] \arrow[r, "(b^2)^*"'] & \omega
\end{tikzcd}$$

We can now eliminate the vertices p, r and o in turn. As a result, we end up with the following trivial generalised automaton:

$$\begin{tikzcd}
{} \arrow[r] & \alpha \arrow[r, "ab(b^2)^*"] & \omega
\end{tikzcd}$$

Thus the language recognised by our original machine is $ab(b^2)^*$. □

We now prove that Algorithm 5.2.4 is correct.

Theorem 5.2.6 *Algorithm 5.2.4 computes a regular expression for the language recognised by a normalised automaton.*

Proof Lemma 5.2.3 tells us that the three operations we apply do not change the language, so we need only prove that the algorithm will always lead to a trivial normalised automaton. Each application of the procedure reduces the number of states by one. In addition, none of the three rules can ever lead to a loop appearing on α or ω. The result is now clear. □

Combining Theorems 5.2.1 and 5.2.6, we have proved the following result.

Theorem 5.2.7 (Kleene) *A language is recognisable iff it is regular.* □

Exercises 5.2

1. For each of the regular expressions below, construct an ε-automaton recognising the corresponding language. The alphabet in question is $A = \{a, b\}$.

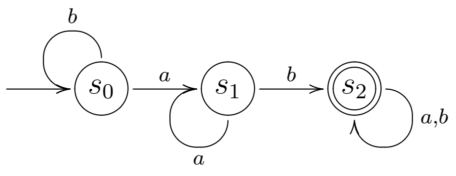
   (i) $a^*(ba^*)^*$.

   (ii) $(a^*b + b^+a)^*$.

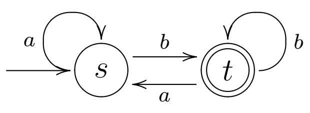
   (iii) $(a^2 + b)^*(a + b^2)^*$.

2. Convert each of the following automata **A** into a normalised automaton and then use Algorithm 5.2.4 to find a regular expression describing $L(\mathbf{A})$.

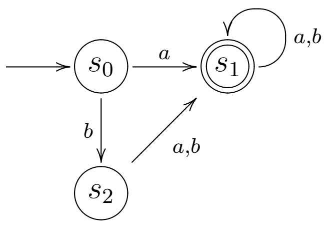
   (i)



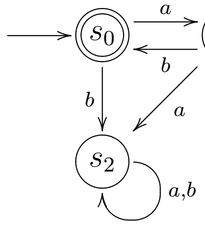
   (ii)



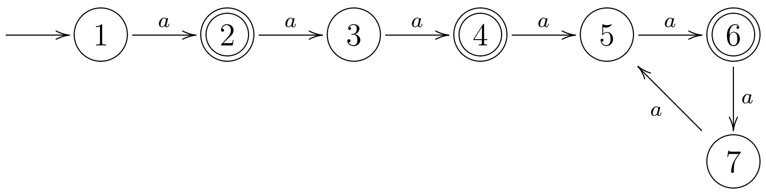
(iii)



(iv)



(v)



3. Prove Lemma 5.2.3.

4. There is an alternative way of proving that the language recognised by an automaton is regular. The questions below give the essential ingredients. First a definition. Let C and R be languages over an alphabet A. A *language equation* is an equation of the form $X = CX + R$. Our goal is to find all solutions to this equation.

   (i) Prove that C^*R is a solution.

   (ii) Prove that if Y is any solution then $C^*R \subseteq Y$.

   (iii) Prove that if $\varepsilon \notin C$ then C^*R is the unique solution. [Hint: let W be any solution and suppose that $W \setminus C^*R$ is not empty. Let z be a string of smallest length in this set. Show that this set contains a still smaller string yielding a contradiction].

5.3 Summary of Chapter 5

- *Regular expressions*: Let A be an alphabet. A regular expression is constructed from the symbols ε, $\emptyset$ and a, where $a \in A$, together with the symbols $+, \cdot$, and * and left and right brackets according to the following rules: ε, $\emptyset$, and a are regular expressions, and if s and t are regular expressions so are $(s + t)$, $(s \cdot t)$ and (s^*).

- *Regular languages*: Every regular expression r describes a language $L(r)$. A language is regular if it can be described by a regular expression.

- *Kleene's theorem*: A language is recognisable if and only if it is regular.

Chapter 6

Minimal automata

We have so far only been concerned with the question of whether or not a language can be recognised by a finite automaton. If it can be, then we have not been interested in how efficiently the job can be done. In this chapter, we shall show that for each recognisable language there is a smallest complete deterministic automaton that recognises it. By 'smallest' we simply mean one having the smallest number of states. As we shall prove later in this section, two deterministic automata that recognise the same language each having the smallest possible number of states must be essentially the same; in mathematical terms, they are isomorphic. This means that with each recognisable language we can associate an automaton that is unique up to isomorphism: this is known as the minimal automaton of the language.

6.1 Partitions and equivalence relations

A collection of individuals can be divided into disjoint groups in many different ways. This simple idea is the main mathematical tool needed in this chapter and forms one of the most important ideas in algebra.

Let X be a set. A *partition* of X is a set P of subsets of X satisfying the following three conditions:

(P1) Each element of P is a non-empty subset of X.

(P2) Distinct elements of P are disjoint.

(P3) Every element X belongs to at least one (and therefore by (P2) exactly one) element of P.

The elements of P are called the *blocks* of the partition.

Examples 6.1.1 Some examples of partitions.

(1) Let

$$X = \{0, 1, \ldots, 9\}$$

and

$$P = \{\{0, 1, 2\}, \{3, 4\}, \{5, 6, 7, 8\}, \{9\}\}.$$

Then P is a partition of X containing four blocks.

(2) The set $\mathbb{N}$ of natural numbers can be partitioned into two blocks: the set of even numbers, and the set of odd numbers.

(3) The set $\mathbb{N}$ can be partitioned into three blocks: those numbers divisible by 3, those numbers that leave remainder 1 when divided by 3, and those numbers that leave remainder 2 when divided by 3.

(4) The set $\mathbb{R}^2$ can be partitioned into infinitely many blocks: consider the set of all lines l_a of the form $y = x + a$ where a is any real number. Each point of $\mathbb{R}^2$ lies on exactly one line of the form l_a.

$\square$

A partition is defined in terms of the set X and the set of blocks P. However, there is an alternative way of presenting this information that is often useful. With each partition P on a set X, we can define a binary relation $\sim_P$ on X as follows:

$$x \sim_P y \Leftrightarrow x \text{ and } y \text{ belong to the same block of } P.$$

The proof of the following is left as an exercise.

Lemma 6.1.2 *The relation $\sim_P$ is reflexive, symmetric, and transitive.* $\square$

Any relation on a set that is reflexive, symmetric, and transitive is called an *equivalence relation.* Thus from each partition we can construct an equivalence relation. In fact, the converse is also true.

Lemma 6.1.3 *Let $\sim$ be an equivalence relation on the set X. For each $x \in X$ put*

$$[x] = \{y \in X \colon x \sim y\}$$

and

$$X/\!\sim\, = \{[x] \colon x \in X\}.$$

Then $X/\!\sim$ is a partition of X.

Proof For each $x \in X$, we have that $x \sim x$, because $\sim$ is reflexive. Thus (P1) and (P3) hold. Suppose that $[x] \cap [y] \neq \emptyset$. Let $z \in [x] \cap [y]$. Then $x \sim z$ and $y \sim z$. By symmetry $z \sim y$, and so by transitivity $x \sim y$. It follows that $[x] = [y]$. Hence (P2) holds. □

The set

$$[x] = \{y \in X \colon x \sim y\}$$

is called the $\sim$-*equivalence class* containing x.

Lemma 6.1.2 tells us how to construct equivalence relations from partitions, and Lemma 6.1.3 tells us how to construct partitions from equivalence relations. The following theorem tells us what happens when we perform these two constructions one after the other.

Theorem 6.1.4 *Let X be a non-empty set.*

(i) *Let P be a partition on X. Then the partition associated with the equivalence relation $\sim_P$ is P.*

(ii) *Let $\sim$ be an equivalence relation on X. Then the equivalence relation associated with the partition $X/\!\sim$ is $\sim$.*

Proof (i) Let P be a partition on X. By Lemma 6.1.2, we can define the equivalence relation $\sim_P$. Let $[x]$ be a $\sim_P$-equivalence class. Then $y \in [x]$ iff $x \sim_P y$ iff x and y are in the same block of P. Thus each $\sim_P$-equivalence class is a block of P. Now let $B \in P$ be a block of P and let $u \in B$. Then $v \in B$ iff $u \sim_P v$ iff $v \in [u]$. Thus $B = [u]$. It follows that each block of P is a $\sim_P$-equivalence class and vice versa. We have shown that P and $X/\!\sim_P$ are the same.

(ii) Let $\sim$ be an equivalence relation on X. By Lemma 6.1.3, we can define a partition $X/\!\sim$ on X. Let $\equiv$ be the equivalence relation defined on X by the partition $X/\!\sim$ according to Lemma 6.1.2. We have that $x \equiv y$ iff

$y \in [x]$ iff $x \sim y$. Thus $\sim$ and $\equiv$ are the same relation. □

Notation Let ρ be an equivalence relation on a set X. Then the ρ-equivalence class containing x is often denoted $\rho(x)$.

Theorem 6.1.4 tells us that partitions on X and equivalence relations on X are two ways of looking at the same thing. In applications, it is the partition itself that is interesting, but checking that we have a partition is usually done indirectly by checking that a relation is an equivalence relation.

The following example introduces some notation that we shall use throughout this chapter.

Example 6.1.5 Let $X = \{1, 2, 3, 4\}$ and let $P = \{\{2\}, \{1, 3\}, \{4\}\}$. Then P is a partition on X. The equivalence relation $\sim$ associated with P can be described by a set of ordered pairs, and these can be conveniently described by a table. The table has rows and columns labelled by the elements of X. Thus each square can be located by means of its co-ordinates: (a, b) means the square in row a and column b. The square (a, b) is marked with $\surd$ if $a \sim b$ and marked with $\times$ otherwise. Strictly speaking we need only mark the squares corresponding to pairs which are $\sim$-related, but I shall use both symbols.

	1	2	3	4
1	$\surd$	$\times$	$\surd$	$\times$
2	$\times$	$\surd$	$\times$	$\times$
3	$\surd$	$\times$	$\surd$	$\times$
4	$\times$	$\times$	$\times$	$\surd$

In fact, this table contains redundant information because if $a \sim b$ then $b \sim a$. It follows that the squares beneath the leading diagonal need not be marked. Thus we obtain

	1	2	3	4
1	$\surd$	$\times$	$\surd$	$\times$
2	$*$	$\surd$	$\times$	$\times$
3	$*$	$*$	$\surd$	$\times$
4	$*$	$*$	$*$	$\surd$

We call this the *table form* of the equivalence relation. □

Exercises 6.1

1. List all equivalence relations on the set $X = \{1, 2, 3, 4\}$ in:
   (i) Partition form.
   (ii) As sets of ordered pairs.
   (iii) In table form.
2. Prove Lemma 6.1.2.

6.2 The indistinguishability relation

In Section 3.1, we described one way of removing unnecessary states from an automaton: the construction of the accessible part of $\mathbf{A}$, denoted $\mathbf{A}^a$, from $\mathbf{A}$. In this section, we shall describe a different way of reducing the number of states in an automaton without changing the language it recognises. On a point of notation: if T is the set of terminal states of a finite automaton, then T' is the set of non-terminal states. Let $\mathbf{A} = (S, A, s_0, \delta, T)$ be an automaton. Two states $s, t \in S$ are said to be *distinguishable* if there exists $x \in A^*$ such that

$$(s \cdot x, t \cdot x) \in (T \times T') \cup (T' \times T).$$

In other words, for some string x, the states $s \cdot x$ and $t \cdot x$ are not both terminal or both non-terminal. The states s and t are said to be *indistinguishable* if they are not distinguishable. This means that for each $x \in A^*$ we have that

$$s \cdot x \in T \Leftrightarrow t \cdot x \in T.$$

Define the relation $\simeq_{\mathbf{A}}$ on the set of states S by

$$s \simeq_{\mathbf{A}} t \Leftrightarrow s \text{ and } t \text{ are indistinguishable.}$$

We call $\simeq_{\mathbf{A}}$ the *indistinguishability relation*. We shall often write $\simeq$ rather than $\simeq_{\mathbf{A}}$ when the machine $\mathbf{A}$ is clear. The relation $\simeq$ will be our main tool in constructing the minimal automaton of a recognisable language. The following result is left as an exercise.

Lemma 6.2.1 *Let* $\mathbf{A}$ *be an automaton. Then the relation* $\simeq_{\mathbf{A}}$ *is an equivalence relation on the set of states of* $\mathbf{A}$. □

The next lemma will be useful in the proof of Theorem 6.2.3.

Lemma 6.2.2 *In an automaton* $\mathbf{A}$ *with set of terminal states* T *the following hold with respect to the indistinguishability relation* $\simeq$.

(i) *If* $s \simeq t$, *then* s *is terminal if and only if* t *is terminal.*

(ii) *If* $s \simeq t$, *then* $s \cdot a \simeq t \cdot a$ *for each letter* a.

Proof (i) Suppose that s is terminal and $s \simeq t$. Then s terminal means that $s \cdot \varepsilon \in T$. But then $t \cdot \varepsilon \in T$, and so $t \in T$. The converse is proved similarly.

(ii) Let $x \in A^*$. Then $(s \cdot a) \cdot x \in T$ precisely when $s \cdot (ax) \in T$. But $s \simeq s'$ and so

$$s \cdot (ax) \in T \Leftrightarrow s' \cdot (ax) \in T.$$

Hence $(s \cdot a) \cdot x \in T$ precisely when $(s' \cdot a) \cdot x \in T$. It follows that $s \cdot a \simeq s' \cdot a$.$\square$

Let $s \in S$ be a state in an automaton $\mathbf{A}$. Then the $\simeq$-equivalence class containing s will be denoted by $[s]$ or sometimes by $[s]_{\mathbf{A}}$. The set of $\simeq$-equivalence classes will be denoted by $S/\simeq$.

It can happen, of course, that each pair of states in an automaton is distinguishable. This is an important case that we single out for a definition. An automaton $\mathbf{A}$ is said to be *reduced* if the relation $\simeq_{\mathbf{A}}$ is equality.

Theorem 6.2.3 (Reduction of an automaton) *Let* $\mathbf{A} = (S, A, s_0, \delta, T)$ *be a finite automaton. Then there is an automaton* $\mathbf{A}/\simeq$, *which is reduced and recognises* $L(\mathbf{A})$. *In addition, if* $\mathbf{A}$ *is accessible then* $\mathbf{A}/\simeq$ *is accessible.*

Proof Define the machine $\mathbf{A}/\simeq$ as follows:

- The set of states is $S/\simeq$.
- The input alphabet is A.
- The initial state is $[s_0]$.
- The set of terminal states is $\{[s]\colon s \in T\}$.
- The transition function is defined by $[s] \cdot a = [s \cdot a]$ for each $a \in A$.

The transition function is well-defined by Lemma 6.2.2(ii). We have therefore proved that $\mathbf{A}/\simeq$ is a well-defined automaton. A simple induction argument shows that $[s] \cdot x = [s \cdot x]$ for each $x \in A^*$.

We can now prove that $\mathbf{A}/\simeq$ is reduced. Let $[s]$ and $[t]$ be a pair of indistinguishable states in $\mathbf{A}/\simeq$. By definition, $[s] \cdot x$ is terminal if and only if $[t] \cdot x$ is terminal for each $x \in A^*$. Thus $[s \cdot x]$ is terminal if and only if $[t \cdot x]$ is terminal. However, by Lemma 6.2.2, $[q]$ is terminal in $\mathbf{A}/\simeq$ precisely when q is terminal in $\mathbf{A}$. It follows that

$$s \cdot x \in T \Leftrightarrow t \cdot x \in T.$$

But this simply means that s and t are indistinguishable in $\mathbf{A}$. Hence $[s] = [t]$, and so $\mathbf{A}/\simeq$ is reduced.

Next we prove that $L(\mathbf{A}/\simeq) = L(\mathbf{A})$. By definition, $x \in L(\mathbf{A}/\simeq)$ precisely when $[s_0] \cdot x$ is terminal. This means that $[s_0 \cdot x]$ is terminal and so $s_0 \cdot x \in T$ by Lemma 6.2.2. Thus

$$x \in L(\mathbf{A}/\simeq) \Leftrightarrow x \in L(\mathbf{A}).$$

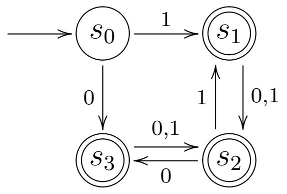
Hence $L(\mathbf{A}/\simeq) = L(\mathbf{A})$.

Finally, we prove that if $\mathbf{A}$ is accessible then $\mathbf{A}/\simeq$ is accessible. Let $[s]$ be a state in $\mathbf{A}/\simeq$. Because $\mathbf{A}$ is accessible there exists $x \in A^*$ such that $s_0 \cdot x = s$. Thus $[s] = [s_0 \cdot x] = [s_0] \cdot x$. It follows that $\mathbf{A}/\simeq$ is accessible. □

We denote the automaton $\mathbf{A}/\simeq$ by $\mathbf{A}^r$ and call it $\mathbf{A}$-*reduced*. For each automaton $\mathbf{A}$, the machine $\mathbf{A}^{ar} = (\mathbf{A}^a)^r$ is both accessible and reduced.

Before we describe an algorithm for constructing $\mathbf{A}^r$, we give an example.

Example 6.2.4 Consider the automaton $\mathbf{A}$ below:



We shall calculate $\simeq$ first, and then $\mathbf{A}/\simeq$ using Theorem 6.2.3. To compute $\simeq$ we shall need to locate the elements of

$$\{s_0, s_1, s_2, s_3\} \times \{s_0, s_1, s_2, s_3\},$$

which belong to $\simeq$. To do this, we shall use the table we described in Example 6.1.5:

	s_0	s_1	s_2	s_3
s_0	$\surd$			
s_1	$*$	$\surd$		
s_2	$*$	$*$	$\surd$	
s_3	$*$	$*$	$*$	$\surd$

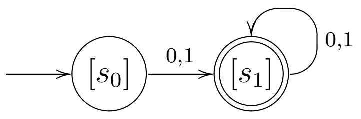
Because each pair of states in $(T \times T') \cup (T' \times T)$ is distinguishable we mark the squares (s_0, s_1), (s_0, s_2) and (s_0, s_3) with a $\times$:

	s_0	s_1	s_2	s_3
s_0	$\surd$	$\times$	$\times$	$\times$
s_1	$*$	$\surd$		
s_2	$*$	$*$	$\surd$	
s_3	$*$	$*$	$*$	$\surd$

To fill in the remaining squares, observe that in this case once the machine reaches the set of terminal states it never leaves it. Thus we obtain the following:

	s_0	s_1	s_2	s_3
s_0	$\surd$	$\times$	$\times$	$\times$
s_1	$*$	$\surd$	$\surd$	$\surd$
s_2	$*$	$*$	$\surd$	$\surd$
s_3	$*$	$*$	$*$	$\surd$

From the table we see that the $\simeq$-equivalence classes are $\{s_0\}$ and $\{s_1, s_2, s_3\}$. We now use the construction described in the proof of Theorem 6.2.3 to construct $\mathbf{A}/\simeq$. This is just



□

We shall now describe an algorithm for constructing $\mathbf{A}^r$.

Remark Before launching into the details of Algorithm 6.2.5 it may be helpful to give a bird's eye view of it. The algorithm in fact determines which pairs are *distinguishable* by marking them with a cross. When the algorithm terminates all the uncrossed pairs are precisely the *indistinguishable* pairs of states which are then marked with ticks. The reason the algorithm works in this way is that it is easier to decide when a pair of states is distinguishable than showing when a pair of states is indistinguishable. The algorithm begins by first crossing the 'obvious' distinguishable pairs of states: namely, those where one state is non-terminal and the other terminal. What makes the body of the algorithm work is the observation that if $(s \cdot a, t \cdot a)$ is distinguishable for some letter a then so too is (s, t).

Algorithm 6.2.5 (Reduction of an automaton) Let **A** be an automaton with set of states $S = \{s_1, \ldots, s_n\}$, initial state s_1, terminal states T, and input alphabet A. The algorithm calculates the equivalence relation $\simeq$. To do so we shall use two tables: table 1 will display the indistinguishability relation at the end of the algorithm, and table 2 will be used for side calculations.

(1) Initialisation: draw up a table (table 1) with rows and columns labelled by the elements of S. Mark main diagonal squares with $\surd$, and squares below the main diagonal with $*$. Mark with $\times$ all squares above the main diagonal in $(T \times T') \cup (T' \times T)$. Squares above the diagonal, which contain neither $\times$ nor $\surd$, are said to be 'empty.'

(2) Main procedure: construct an auxiliary table (table 2) as follows: working from left to right and top to bottom of table 1, label each row of table 2 with the pair (s, t) whenever the (s, t)-entry in table 1 is empty; the columns are labelled by the elements of A.

Now work from top to bottom of table 2: for each pair (s, t) labelling a row calculate the states $(s \cdot a, t \cdot a)$ for each $a \in A$ and enter them in table 2:

- If any of these pairs of states or $(t \cdot a, s \cdot a)$ labels a square marked with a $\times$ in table 1 then mark (s, t) with a $\times$ in table 1.
- If all the pairs $(s \cdot a, t \cdot a)$ are diagonal, mark (s, t) with a $\surd$ in table 1.
- Otherwise do not mark (s, t) and move to the next row.

(3) Finishing off: work from left to right and top to bottom of table 1. For each empty square (s, t) use table 2 to find all the squares $(s \cdot a, t \cdot a)$:

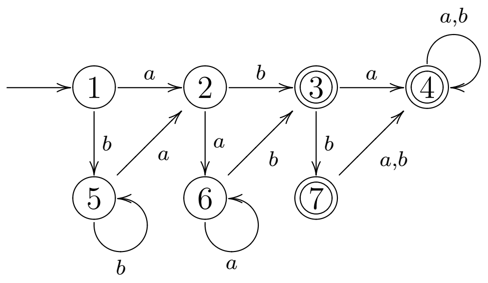
- If any of these squares in table 1 contains $\times$, then mark (s, t) with a $\times$ in table 1 and move to the next empty square.
- If all of these squares in table 1 contain $\surd$, then mark (s, t) with $\surd$ in table 1 and move to the next empty square.
- In all other cases move to the next empty square.

When an iteration of this procedure is completed we say that a 'pass' of table 1 has been completed. This procedure is repeated until a pass occurs in which no new squares are marked with $\times$, or until there are no empty squares. At this point, all empty squares are marked with $\surd$ and the algorithm terminates.

□

Before we prove that the algorithm works, we give an example.

Example 6.2.6 Consider the automaton **A** below:



We shall use the algorithm to compute $\simeq$. The first step is to draw up the initialised table 1:

	1	2	3	4	5	6	7
1	$\surd$		$\times$	$\times$			$\times$
2	$*$	$\surd$	$\times$	$\times$			$\times$
3	$*$	$*$	$\surd$		$\times$	$\times$	
4	$*$	$*$	$*$	$\surd$	$\times$	$\times$	
5	$*$	$*$	$*$	$*$	$\surd$		$\times$
6	$*$	$*$	$*$	$*$	$*$	$\surd$	$\times$
7	$*$	$*$	$*$	$*$	$*$	$*$	$\surd$

We now construct table 2 and at the same time modify table 1:

	a	b
$(1,2)$	$(2,6)$	$(5,3)$
$(1,5)$	$(2,2)$	$(5,5)$
$(1,6)$	$(2,6)$	$(5,3)$
$(2,5)$	$(6,2)$	$(3,5)$
$(2,6)$	$(6,6)$	$(3,3)$
$(3,4)$	$(4,4)$	$(7,4)$
$(3,7)$	$(4,4)$	$(7,4)$
$(4,7)$	$(4,4)$	$(4,4)$
$(5,6)$	$(2,6)$	$(5,3)$

As a result the squares,

$$(1,2),(1,6),(2,5),(5,6),$$

are all marked with $\times$ in table 1, whereas the squares,

$$(1,5),(2,6),(4,7),$$

are marked with $\surd$. The squares,

$$(3,4),(3,7),$$

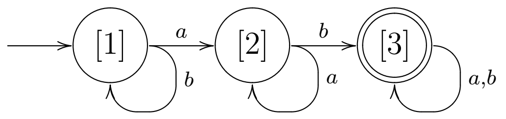
are left unchanged. Table 1 now has the following form:

	1	2	3	4	5	6	7
1	$\surd$	$\times$	$\times$	$\times$	$\surd$	$\times$	$\times$
2	$*$	$\surd$	$\times$	$\times$	$\times$	$\surd$	$\times$
3	$*$	$*$	$\surd$		$\times$	$\times$	
4	$*$	$*$	$*$	$\surd$	$\times$	$\times$	$\surd$
5	$*$	$*$	$*$	$*$	$\surd$	$\times$	$\times$
6	$*$	$*$	$*$	$*$	$*$	$\surd$	$\times$
7	$*$	$*$	$*$	$*$	$*$	$*$	$\surd$

To finish off, we check each empty square (s,t) in table 1 in turn to see if the corresponding entries in table 2 should cause us to mark this square. When we do this we find that no squares are changed. Thus the algorithm terminates. We now place $\surd$'s in all blank squares. We arrive at the following table:

	1	2	3	4	5	6	7
1	$\surd$	$\times$	$\times$	$\times$	$\surd$	$\times$	$\times$
2	$*$	$\surd$	$\times$	$\times$	$\times$	$\surd$	$\times$
3	$*$	$*$	$\surd$	$\surd$	$\times$	$\times$	$\surd$
4	$*$	$*$	$*$	$\surd$	$\times$	$\times$	$\surd$
5	$*$	$*$	$*$	$*$	$\surd$	$\times$	$\times$
6	$*$	$*$	$*$	$*$	$*$	$\surd$	$\times$
7	$*$	$*$	$*$	$*$	$*$	$*$	$\surd$

We can read off the $\simeq$-equivalence classes from this table. They are $\{1,5\}$, $\{2,6\}$ and $\{3,4,7\}$. The automaton $\mathbf{A}/\simeq$ is therefore



□

We now justify that this algorithm works.

Theorem 6.2.7 *Algorithm 6.2.5 is correct*

Proof Let $\mathbf{A} = (S, A, s_0, \delta, T)$ be an automaton. By definition, the pair of states (s,t) is distinguishable if and only if there is a string $x \in A^*$ such that

$$(s \cdot x, t \cdot x) \in (T \times T') \cup (T' \times T);$$

I shall say that x *distinguishes* s and t. Those states distinguished by the empty string are precisely the elements of

$$(T \times T') \cup (T' \times T).$$

Suppose that (s,t) is distinguished by a string y of length $n > 0$. Put $y = ax$ where $a \in A$ and $x \in A^*$. Then $(s \cdot a, t \cdot a)$ is distinguished by the string x of length $n-1$. It follows that the pair (s,t) is distinguishable if and only if there is a sequence of pairs of states,

$$(s_0, t_0), (s_1, t_1), \ldots, (s_n, t_n),$$

such that $(s,t) = (s_0, t_0)$, and $(s_n, t_n) \in (T \times T') \cup (T' \times T)$ and

$$(s_i, t_i) = (s_{i-1} \cdot a_i, t_{i-1} \cdot a_i)$$

for $1 \leq i \leq n$ for some $a_i \in A$. The algorithm marks the pairs of states in $(T \times T') \cup (T' \times T)$ with a cross, and marks (s,t) with a cross whenever the square $(s \cdot a, t \cdot a)$ (or the square $(t \cdot a, s \cdot a)$) is marked with a cross for some $a \in A$. It is now clear that if the algorithm marks a square (s,t) with a cross, then s and t are distinguishable.

It therefore remains to prove that if a pair of states is distinguishable, then the corresponding square (or the appropriate one above the diagonal) is marked with a cross by the algorithm. We shall prove this by induction on the length of the strings that distinguish the pair. If the pair can be distinguished by the empty string then the corresponding square will be marked with a cross during initialisation. Suppose now that the square corresponding to any pair of states that can be distinguished by a string of length n is marked with a cross by the algorithm. Let (s,t) be a pair of states that can be distinguished by a string y of length $n+1$. Let $y = ax$ where $a \in A$ and x has length n. Then the pair $(s \cdot a, t \cdot a)$ can be distinguished by the string x, which has length n. By the induction hypothesis, the square $(s \cdot a, t \cdot a)$ will be marked with a cross by the algorithm. But then the square (s,t) will be marked with a cross either during the main procedure or whilst finishing off. $\square$

Exercises 6.2

1. Let $\mathbf{A}$ be a finite automaton. Prove Lemma 6.2.1 that $\simeq_{\mathbf{A}}$ is an equivalence relation on the set of states of $\mathbf{A}$.

2. Complete the proof of Theorem 6.2.3, by showing that $[s] \cdot x = [s \cdot x]$ for each $x \in A^*$.

3. For each of the automata $\mathbf{A}$ below find $\mathbf{A}^r$. In each case, we present the automaton by means of its transition table turned on its side. This helps in the calculations.

   (i)

1	2	3	4	5	6	7	
2	2	5	6	5	6	5	a
4	3	3	4	7	7	7	b

   The initial state is 1 and the terminal states are $3, 5, 6, 7$.

(ii)

0	1	2	3	4	5	
1	3	4	5	5	2	a
2	4	3	0	0	2	b

The initial state is 0 and the terminal states are 0 and 5.

(iii)

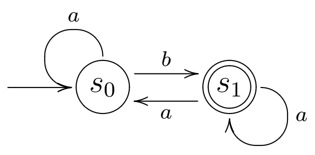
1	2	3	4	5	6	7	8	
2	7	1	3	8	3	7	7	a
6	3	3	7	6	7	5	3	b

The initial state is 1 and the terminal state is 3.

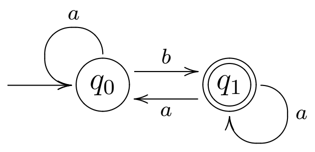
4. Let $\mathbf{A} = (S, A, i, \delta, \{t\})$ be an automaton with exactly one terminal state, and the property that for each $s \in S$ there is a string $x \in A^*$ such that $s \cdot x = t$. Suppose that for each $a \in A$ the function τ_a, defined by τ_a maps s to $s \cdot a$ for each $s \in S$, is a bijection. Prove that $\mathbf{A}$ is reduced.

6.3 Isomorphisms of automata

We begin with an example. Consider the following two automata, which we denote by $\mathbf{A}$ and $\mathbf{B}$, respectively:



and



These automata are different because the labels on the states are different. But in every other respect, $\mathbf{A}$ and $\mathbf{B}$ are 'essentially the same.' In this case, it was easy to see that the two automata were essentially the same, but if they each had more states then it would have been much harder. In order to realise the main goal of this chapter, we need to have a precise mathematical definition of when two automata are essentially the same, one that we can check in a systematic way however large the automata involved. The definition below provides the answer to this question.

Let $\mathbf{A} = (S, A, s_0, \delta, F)$ and $\mathbf{B} = (Q, A, q_0, \gamma, G)$ be two automata with the same input alphabet A. An *isomorphism* θ *from* $\mathbf{A}$ *to* $\mathbf{B}$, denoted by $\theta\colon \mathbf{A} \to \mathbf{B}$, is a function $\theta\colon S \to Q$ satisfying the following four conditions:

(IM1) The function θ is bijective.

(IM2) $\theta(s_0) = q_0$.

(IM3) $s \in F \Leftrightarrow \theta(s) \in G$.

(IM4) $\theta(\delta(s, a)) = \gamma(\theta(s), a)$ for each $s \in S$ and $a \in A$.

If we use our usual notation for the transition function in an automaton, then (IM4) would be written as

$$\theta(s \cdot a) = \theta(s) \cdot a.$$

If there is an isomorphism from $\mathbf{A}$ to $\mathbf{B}$ we say that $\mathbf{A}$ is *isomorphic* to $\mathbf{B}$, denoted by $\mathbf{A} \equiv \mathbf{B}$. Isomorphic automata may differ in their state labelling and may look different when drawn as directed graphs, but by suitable relabelling, and by moving states and bending transitions, they can be made to look identical.

Lemma 6.3.1 *Let* $\mathbf{A} = (S, A, s_0, \delta, F)$ *and* $\mathbf{B} = (Q, A, q_0, \gamma, G)$ *be automata, and let* $\theta\colon \mathbf{A} \to \mathbf{B}$ *be an isomorphism. Then*

$$\theta(\delta^*(s, x)) = \gamma^*(\theta(s), x)$$

for each $s \in S$ *and* $x \in A^*$. *In particular,* $L(\mathbf{A}) = L(\mathbf{B})$.

Proof Using our usual notation for the extended state transition function, the lemma states that

$$\theta(s \cdot x) = \theta(s) \cdot x.$$

We prove the first assertion by induction on the length of x. Base step: we check the result holds when $x = \varepsilon$:

$$\theta(s \cdot \varepsilon) = \theta(s) \text{ whereas } \theta(s) \cdot \varepsilon = \theta(s),$$

as required. Induction hypothesis: assume the result holds for all strings of length at most n. Let u be a string of length $n+1$. Then $u = ax$ where $a \in A$ and x has length n. Now

$$\theta(s \cdot u) = \theta(s \cdot (ax)) = \theta((s \cdot a) \cdot x).$$

Put $s' = s \cdot a$. Then

$$\theta((s \cdot a) \cdot x) = \theta(s' \cdot x) = \theta(s') \cdot x$$

by the induction hypothesis. However,

$$\theta(s') = \theta(s \cdot a) = \theta(s) \cdot a$$

by (IM4). Hence

$$\theta(s \cdot u) = (\theta(s) \cdot a) \cdot x = \theta(s) \cdot (ax) = \theta(s) \cdot u,$$

as required.

We now prove that $L(\mathbf{A}) = L(\mathbf{B})$. By definition

$$x \in L(\mathbf{A}) \Leftrightarrow s_0 \cdot x \in F.$$

By (IM3),

$$s_0 \cdot x \in F \Leftrightarrow \theta(s_0 \cdot x) \in G.$$

By our result above,

$$\theta(s_0 \cdot x) = \theta(s_0) \cdot x.$$

By (IM2), we have that $\theta(s_0) = q_0$, and so

$$s_0 \cdot x \in F \Leftrightarrow q_0 \cdot x \in G.$$

Hence $x \in L(\mathbf{A})$ if and only if $x \in L(\mathbf{B})$ and so $L(\mathbf{A}) = L(\mathbf{B})$ as required.$\square$

Exercises 6.3

1. Let **A**, **B** and **C** be automata. Prove the following:

   (i) $\mathbf{A} \equiv \mathbf{A}$; each automaton is isomorphic to itself.

   (ii) If $\mathbf{A} \equiv \mathbf{B}$ then $\mathbf{B} \equiv \mathbf{A}$; if **A** is isomorphic to **B** then **B** is isomorphic to **A**.

   (iii) If $\mathbf{A} \equiv \mathbf{B}$ and $\mathbf{B} \equiv \mathbf{C}$ then $\mathbf{A} \equiv \mathbf{C}$; if $\mathbf{A}$ is isomorphic to $\mathbf{B}$, and $\mathbf{B}$ is isomorphic to $\mathbf{C}$ then $\mathbf{A}$ is isomorphic to $\mathbf{C}$.

2. Let θ: $\mathbf{A} \to \mathbf{B}$ be an isomorphism from $\mathbf{A} = (S, A, s_0, \delta, F)$ to $\mathbf{B} = (Q, A, q_0, \gamma, G)$. Prove that:

   (i) The number of states of $\mathbf{A}$ is the same as the number of states of $\mathbf{B}$.

   (ii) The number of terminal states of $\mathbf{A}$ is the same as the number of terminal states of $\mathbf{B}$.

   (iii) $\mathbf{A}$ is accessible if and only if $\mathbf{B}$ is accessible.

   (iv) $\mathbf{A}$ is reduced if and only if $\mathbf{B}$ is reduced.

3. Let $\mathbf{A}$ be an accessible automaton. Show that if θ, ϕ: $\mathbf{A} \to \mathbf{B}$ are both isomorphisms then $\theta = \phi$.

6.4 The minimal automaton

We now come to a fundamental definition. Let L be a recognisable language. A complete deterministic automaton $\mathbf{A}$ is said to be *minimal (for L)* if $L(\mathbf{A}) = L$ and if $\mathbf{B}$ is any complete deterministic automaton such that $L(\mathbf{B}) = L$, then the number of states of $\mathbf{A}$ is less than or equal to the number of states of $\mathbf{B}$. Minimal automata for a language L certainly exist. The problem is to find a way of constructing them. Our first result narrows down the search.

Lemma 6.4.1 *Let L be a recognisable language. If $\mathbf{A}$ is minimal for L, then $\mathbf{A}$ is both accessible and reduced.*

Proof If $\mathbf{A}$ is not accessible, then $\mathbf{A}^a$ has fewer states than $\mathbf{A}$ and $L(\mathbf{A}^a) = L$. But this contradicts the definition of $\mathbf{A}$. It follows that $\mathbf{A}$ is accessible. A similar argument shows that $\mathbf{A}$ is reduced. $\square$

If $\mathbf{A}$ is minimal for L, then $\mathbf{A}$ must be both reduced and accessible. The next result tells us that any reduced accessible automaton recognising L is in fact minimal.

Theorem 6.4.2 *Let L be a recognisable language.*

(i) *Any two reduced accessible automata recognising L are isomorphic.*

(ii) *Any reduced accessible automaton recognising L is a minimal automaton for L.*

(iii) *Any two minimal automata for L are isomorphic.*

Proof (i) Let $\mathbf{A} = (S, A, s_0, \delta, F)$ and $\mathbf{B} = (Q, A, q_0, \gamma, G)$ be two reduced accessible automata such that $L(\mathbf{A}) = L(\mathbf{B})$. We prove that $\mathbf{A}$ is isomorphic to $\mathbf{B}$. To do this, we have to conjure up an isomorphism from $\mathbf{A}$ to $\mathbf{B}$. To keep the notation simple, we shall use the 'dot' notation for both δ^* and γ^*. We shall use the following observation:

$$s_0 \cdot x \in F \quad \Leftrightarrow \quad q_0 \cdot x \in G, \tag{6.1}$$

which follows from the fact that $L(\mathbf{A}) = L(\mathbf{B})$.

Let $s \in S$. Because $\mathbf{A}$ is accessible there exists $x \in A^*$ such that $s = s_0 \cdot x$. Define

$$\theta(s) = q_0 \cdot x.$$

To show that θ is a well-defined injective function we have to prove that

$$s_0 \cdot x = s_0 \cdot y \Leftrightarrow q_0 \cdot x = q_0 \cdot y.$$

Now $\mathbf{B}$ is reduced, so it will be enough to prove that

$$s_0 \cdot x \simeq_{\mathbf{A}} s_0 \cdot y \Leftrightarrow q_0 \cdot x \simeq_{\mathbf{B}} q_0 \cdot y.$$

Now $s_0 \cdot x \simeq_{\mathbf{A}} s_0 \cdot y$ iff for all $w \in A^*$:

$$(s_0 \cdot x) \cdot w \in F \Leftrightarrow (s_0 \cdot y) \cdot w \in F.$$

This is equivalent to

$$s_0 \cdot (xw) \in F \Leftrightarrow s_0 \cdot (yw) \in F.$$

By (6.1) above this is equivalent to

$$q_0 \cdot (xw) \in G \Leftrightarrow q_0 \cdot (yw) \in G.$$

Finally, we deduce that $q_0 \cdot x \simeq_{\mathbf{B}} q_0 \cdot y$. We have therefore proved that θ is well-defined injective function.

To show that θ is surjective, let q be an arbitrary state in $\mathbf{B}$. By assumption, $\mathbf{B}$ is accessible and so there exists $x \in A^*$ such that $q = q_0 \cdot x$. Put $s = s_0 \cdot x$ in $\mathbf{A}$. Then by definition $\theta(s) = q$, and so θ is surjective as required. We have therefore proved that (IM1) holds.

That (IM2) holds is immediate because $s_0 = s_0 \cdot \varepsilon$. Thus $\theta(s_0) = q_0 \cdot \varepsilon = q_0$ as required.

(IM3) holds by accessibility and (6.1).

(IM4) holds: for each $s \in S$ and $a \in A$ we have to prove that $\theta(s \cdot a) = \theta(s) \cdot a$. Let $s = s_0 \cdot x$ for some $x \in A^*$. Then $\theta(s) = q_0 \cdot x$. Thus

$$\theta(s) \cdot a = (q_0 \cdot x) \cdot a = q_0 \cdot (xa).$$

On the other hand,

$$s \cdot a = (s_0 \cdot x) \cdot a = s_0 \cdot (xa).$$

Hence by definition,

$$\theta(s \cdot a) = q_0 \cdot (xa) = \theta(s) \cdot a$$

and the result follows.

(ii) Let $\mathbf{A}$ be a reduced and accessible automaton recognising L. We prove that $\mathbf{A}$ is minimal for L. Let $\mathbf{B}$ be any automaton recognising L. Then $L = L(\mathbf{B}^{ar})$ and the number of states in $\mathbf{B}^{ar}$ is less than or equal to the number of states in $\mathbf{B}$. But by (i), $\mathbf{A}$ and $\mathbf{B}^{ar}$ are isomorphic and so, in particular, have the same number of states. It follows that the number of states in $\mathbf{A}$ is less than or equal to the number of states in $\mathbf{B}$. Thus $\mathbf{A}$ is a minimal automaton for L.

(iii) By Lemma 6.4.1, a minimal automaton for L is accessible and reduced. By (i), any two accessible and reduced automata recognising L are isomorphic. Thus any two minimal automata for a language are isomorphic.$\square$

We can paraphrase the above theorem in the following way: the minimal automaton for a recognisable language is *unique up to isomorphism*. Because of this we shall often refer to *the* minimal automaton of a recognisable language. The number of states in a minimal automaton for a recognisable language L is called the *rank* of the language L. This can be regarded as a measure of the complexity of L.

Observe that if $\mathbf{A}$ is an automaton, then $\mathbf{A}^{ar}$ and $\mathbf{A}^{ra}$ are both reduced and accessible and recognise $L(\mathbf{A})$. So in principle, we could calculate either of these two automata to find the mimimal automaton. However, it makes sense to compute $\mathbf{A}^{ar} = (\mathbf{A}^a)^r$ rather than $\mathbf{A}^{ra}$. This is because calculating the reduction of an automaton is more labour intensive than calculating the accessible part. By calculating $\mathbf{A}^a$ first, we will in general reduce the number of states and so decrease the amount of work needed in the subsequent reduction.

Algorithm 6.4.3 (Minimal automaton) This algorithm computes the minimal automaton for a recognisable language L from any complete deterministic automaton $\mathbf{A}$ recognising L. Calculate $\mathbf{A}^a$, the accessible part of $\mathbf{A}$, using Algorithm 3.1.4. Next calculate the reduction of $\mathbf{A}^a$, using Algorithm 6.2.5. The automaton $\mathbf{A}^{ar}$ that results is the minimal automaton for L. □

Exercises 6.4

1. Find the rank of each subset of $(0 + 1)^2$. You should first list all the subsets; construct deterministic automata that recognise each subset; and finally, convert your automata to minimal automata.

2. Let $n \geq 2$. Define
$$L_n = \{x \in (a + b)^*\colon |\, x \,| \equiv 0 \pmod{n}\}.$$
Prove that the rank of L_n is n.

3. Determine the rank of $(0 + 1)^*1(0 + 1)^{n-1}$ for $n \geq 1$. *This question refers back to Exercises 3.2, Question 3.*

4. Let $\mathbf{A} = (S, A, s_0, \delta, F)$ and $\mathbf{B} = (Q, A, q_0, \gamma, G)$ be complete deterministic automata. A *homomorphism* θ from $\mathbf{A}$ to $\mathbf{B}$ is a function $\theta\colon S \to Q$ such that $\theta(s_0) = q_0$, $\theta(s \cdot a) = \theta(s) \cdot a$, where a is an input letter, and if $s \in F$ then $\theta(s) \in G$.

   (i) Prove by induction that $\theta(s \cdot x) = \theta(s) \cdot x$ for all strings x.

   (ii) Prove that if there is a homomorphism from $\mathbf{A}$ to $\mathbf{B}$ then $L(\mathbf{A}) \subseteq L(\mathbf{B})$.

   (iii) Prove that if $\mathbf{A}$ is accessible then there is at most one homomorphism from $\mathbf{A}$ to $\mathbf{B}$.

   (iv) Prove that a bijective homomorphism is an isomorphism.

   (v) Prove that if $\mathbf{A}$ is accessible, $\mathbf{B}$ is reduced and $L(\mathbf{A}) = L(\mathbf{B})$ then there is a homomorphism from $\mathbf{A}$ to $\mathbf{B}$.

6.5 The method of quotients

In Section 6.4, we showed that if $L = L(\mathbf{A})$ then the minimal automaton for L is $\mathbf{A}^{ar}$. In this section, we shall construct the minimal automaton of L directly from a regular expression for L. Our method is based on a new language operation.

Let L be a language over the alphabet A and let $u \in A^*$. Define the *left quotient of* L *by* u to be

$$u^{-1}L = \{v \in A^*\colon uv \in L\}.$$

The notation is intended to help you remember the meaning:

$$v \in u^{-1}L \Leftrightarrow uv \in uu^{-1}L \Leftrightarrow uv \in L,$$

because we think of u as being cancelled by u^{-1}.

Terminology In this section, I shall deal only with left quotients, so when I write 'quotient,' I shall always mean 'left quotient.'

Examples 6.5.1 Let A be an alphabet, $a \in A$ and L a language over A.

(1) $a^{-1}a = \varepsilon$. Remember that $a^{-1}a$ means $a^{-1}\{a\}$. By definition $u \in a^{-1}\{a\}$ iff $au \in \{a\}$. Thus $au = a$ and so $u = \varepsilon$. It follows that $a^{-1}a = \varepsilon$.

(2) $a^{-1}\varepsilon = \emptyset$. Let $u \in a^{-1}\{\varepsilon\}$. Then $au \in \{\varepsilon\}$ and so $au = \varepsilon$. However there is no string u which satisfies this condition. Consequently $a^{-1}\varepsilon = \emptyset$.

(3) $a^{-1}\emptyset = \emptyset$. This is proved by a similar argument to that in (2) above.

(4) $a^{-1}b = \emptyset$ if $b \in A$ and $b \neq a$. Let $u \in a^{-1}\{b\}$. Then $au = b$. There are no solutions to this equation and so $a^{-1}b = \emptyset$.

(5) $\varepsilon^{-1}L = L$. By definition $u \in \varepsilon^{-1}L$ iff $\varepsilon u \in L$. This just means that $u \in L$. Hence $\varepsilon^{-1}L = L$.

□

The quotients of a regular language, as we shall show, can be used to construct the minimal automaton of the language. So we shall need to develop ways of computing quotients efficiently. To do this, the following simple definition will be invaluable. Let L be any language. Define

$$\delta(L) = \begin{cases} \emptyset & \text{if } \varepsilon \notin L \\ \{\varepsilon\} & \text{if } \varepsilon \in L. \end{cases}$$

Thus $\delta(L)$ simply records the absence or presence of ε in the language. The following lemma provides the tools necessary for computing δ for any language given by means of a regular expression. The proofs are straightforward and left as exercises.

Lemma 6.5.2 *Let A be an alphabet and $L, M \subseteq A^*$.*

(i) $\delta(a) = \emptyset$ *for each* $a \in A$.

(ii) $\delta(\emptyset) = \emptyset$.

(iii) $\delta(\varepsilon) = \varepsilon$.

(iv) $\delta(LM) = \delta(L) \cap \delta(M)$.

(v) $\delta(L + M) = \delta(L) + \delta(M)$.

(vi) $\delta(L^*) = \varepsilon$.

□

We now show how to compute quotients.

Proposition 6.5.3 *Let $u, v \in A^*$ and $a \in A$.*

(i) *If* $L = \emptyset$ *or* ε *then* $u^{-1}(LM) = L(u^{-1}M)$.

(ii) *If* $\{L_i : i \in I\}$ *is any family of languages then* $u^{-1}(\sum_{i \in I} L_i) = \sum_{i \in I} u^{-1} L_i$.

(iii) $a^{-1}(LM) = (a^{-1}L)M + \delta(L)(a^{-1}M)$.

(iv) $a^{-1}L^* = (a^{-1}L)L^*$.

(v) $(uv)^{-1}L = v^{-1}(u^{-1}L)$.

Proof (i) Straightforward.

(ii) By definition $v \in u^{-1}(\sum_{i\in I} L_i)$ iff $uv \in \sum_{i\in I} L_i$. But $uv \in \sum_{i\in I} L_i$ implies $uv \in L_i$ for some $i \in I$. Thus $v \in u^{-1}L_i$ for some $i \in I$. It follows that $v \in \sum_{i\in I} u^{-1}L_i$. The converse is proved similarly.

(iii) Write $L = \delta(L) + L_0$ where $L_0 = L \setminus \varepsilon$. Then

$$a^{-1}(LM) = a^{-1}(\delta(L)M + L_0M) = \delta(L)(a^{-1}M) + a^{-1}(L_0M),$$

using (i) and (ii). It is therefore enough to prove the result for the case where L does not contain ε. We have to prove that

$$a^{-1}(LM) = (a^{-1}L)M$$

if $\varepsilon \notin L$. Let $x \in a^{-1}(LM)$. Then $ax = lm$ where $l \in L$ and $m \in M$ and $l \neq \varepsilon$, by assumption. Thus $l = al'$ for some l'. It follows that $x = l'm$. Also $l = al' \in L$ iff $l' \in a^{-1}L$. Thus $x \in (a^{-1}L)M$. Conversely, if $x \in (a^{-1}L)M$, then $x = l'm$ for some $l' \in a^{-1}L$ and $m \in M$. But then $al' \in L$ and so $ax = (al')m \in LM$.

(iv) By definition $x \in a^{-1}L^*$ iff $ax \in L^*$. Thus $ax = u_1 \ldots u_n$ for some non-empty $u_i \in L$. Now $u_1 = au$ for some u. Hence $x = u(u_2 \ldots u_n)$, where $u \in a^{-1}L$. Thus $x \in (a^{-1}L)L^*$. Conversely, if $x \in (a^{-1}L)L^*$ then $x = u(u_2 \ldots u_n)$ for some $u \in a^{-1}L$. It follows that $au \in L$ and so $ax \in L^*$. Hence $x \in a^{-1}L^*$.

(v) By definition $x \in (uv)^{-1}L$ iff $(uv)x \in L$ iff $u(vx) \in L$ iff $vx \in u^{-1}L$ iff $x \in v^{-1}(u^{-1}L)$. Hence $(uv)^{-1}L = v^{-1}(u^{-1}L)$. □

It is important to note that in parts (iii) and (iv) above we have derived expressions for quotients by means of a *single letter only*.

Examples 6.5.4 In the examples below, $A = \{a, b\}$.

(1) $a^{-1}A = \{\varepsilon\}$. We can write $A = a + b$. Thus

$$a^{-1}A = a^{-1}(a + b) = a^{-1}a + a^{-1}b = \varepsilon + \emptyset = \varepsilon.$$

(2) $a^{-1}A^* = A^* = b^{-1}A^*$. This is straightforward.

(3) Let x be a non-empty string that does not begin with a. Then $a^{-1}(xA^*) = \emptyset$. This is because $y \in a^{-1}(xA^*)$ iff $ay \in xA^*$. But x does not begin with a. So there is no solution for y.

(4) $a^{-1}(axA^*) = xA^*$. This is because $y \in a^{-1}(axA^*)$ iff $ay \in axA^*$. This can only be true if $y \in xA^*$.

(5) Calculate $a^{-1}(A^*abaA^*)$. We can regard A^*abaA^* as a product of two languages in a number or ways, any one of which can be chosen. We choose to regard it as A^* followed by $abaA^*$. Thus by Proposition 6.5.3(iii), we have that

$$a^{-1}(A^*abaA^*) = (a^{-1}A^*)(abaA^*) + \delta(A^*)a^{-1}(abaA^*).$$

We have already shown that $a^{-1}A^* = A^*$ and that $a^{-1}(abaA^*) = baA^*$. Thus

$$a^{-1}(A^*abaA^*) = A^*abaA^* + baA^*.$$

□

We now prove two important results.

Proposition 6.5.5

(i) *The left quotient of a recognisable language is recognisable.*

(ii) *A recognisable language has only a finite number of distinct left quotients.*

Proof (i) Let L be a recognisable language. Then $L = L(\mathbf{A})$ where $\mathbf{A} = (S, A, i, \delta, T)$ is an automaton. We prove first that every left quotient of L is recognisable. Let $u \in A^*$ and put $i' = i \cdot u$. Put $\mathbf{A}_u = (S, A, i', \delta, T)$. We claim that $L(\mathbf{A}_u) = u^{-1}L$. Let $x \in u^{-1}L$. Then $ux \in L$. Thus $i \cdot (ux) \in T$ and so $(i \cdot u) \cdot x \in T$. Hence $i' \cdot x \in T$ giving $x \in \mathbf{A}_u$. We have therefore proved that $u^{-1}L \subseteq L(\mathbf{A}_u)$. To prove the reverse inclusion let $x \in L(\mathbf{A}_u)$. Then $i' \cdot x \in T$ and so $(i \cdot u) \cdot x \in T$. This means that $i \cdot (ux) \in T$ and so $ux \in L(\mathbf{A}) = L$. Hence $x \in u^{-1}L$, as required.

(ii) To finish off, we have to prove that there are only finitely many left quotients. The set of left quotients of L is just the set of languages $L(\mathbf{A}_s)$, where $\mathbf{A}_s = (S, A, s, \delta, T)$ and $s \in S$, and there are clearly only a finite number of these. □

We can also prove the converse of the above result.

Proposition 6.5.6 *Let L be a language with only a finite number of distinct left quotients. Then L is recognisable.*

Proof We shall construct a finite automaton $\mathbf{A}_L = (S, A, i, \delta, T)$ such that $L(\mathbf{A}) = L$. Define

- $S = \{u^{-1}L \colon u \in A^*\}$, which is finite by assumption.
- $i = L = \varepsilon^{-1}L$.
- $T = \{u^{-1}L \colon \varepsilon \in u^{-1}L\}$; those quotients of L which contain ε.
- $\delta(u^{-1}L, a) = a^{-1}(u^{-1}L) = (ua)^{-1}L$, using Proposition 6.5.3(v).

By construction, $\mathbf{A}_L$ is a complete deterministic automaton. To calculate $L(\mathbf{A}_L)$ we need to determine δ^*. We claim that

$$\delta^*(u^{-1}L, x) = (ux)^{-1}L$$

for each $x \in A^*$. We leave the proof of this as an exercise.

By definition, $w \in L(\mathbf{A}_L)$ iff $\delta^*(i, w) \in T$ iff $\delta^*(L, w) \in T$. From the form of δ^* and the definition of T this is equivalent to $\varepsilon \in w^{-1}L$, which means precisely that $w \in L$. Hence $L(\mathbf{A}_L) = L$. $\square$

Combining Propositions 6.5.5 and 6.5.6, we now have the following new characterisation of recognisable languages.

Theorem 6.5.7 *A language is recognisable if and only if it has a finite number of distinct left quotients.* $\square$

The automaton $\mathbf{A}_L$ constructed from a recognisable language L in Proposition 6.5.6 is the best we can hope for.

Theorem 6.5.8 *Let L be a recognisable language. Then $\mathbf{A}_L$ is the minimal automaton of L.*

Proof By Theorem 6.4.2, it is enough to show that $\mathbf{A}_L$ is reduced and accessible. The proof that $\mathbf{A}_L$ is accessible is almost immediate: let $u^{-1}L$ be an arbitrary state in $\mathbf{A}_L$. Then $\delta^*(L, u) = u^{-1}L$ and L is the initial state and so $\mathbf{A}_L$ is accessible. To prove that $\mathbf{A}_L$ is reduced, suppose that $u^{-1}L \simeq v^{-1}L$. Then by definition, for each $x \in A^*$ we have that

$$\delta^*(u^{-1}L, x) \in T \Leftrightarrow \delta^*(v^{-1}L, x) \in T.$$

This is equivalent to saying that for each $x \in A^*$, we have that

$$\varepsilon \in (ux)^{-1}L \Leftrightarrow \varepsilon \in (vx)^{-1}L.$$

In other words, $x \in u^{-1}L \Leftrightarrow x \in v^{-1}L$. Hence $u^{-1}L = v^{-1}L$. □

We now describe an algorithm that takes as input a regular expression for a language L and produces as output the minimal automaton $\mathbf{A}_L$. This algorithm has one drawback, which we explain at the end of this section.

Algorithm 6.5.9 (Method of Quotients) Given a regular expression for the recognisable language L, this algorithm constructs the minimal automaton for L. We denote the regular expression describing L also by L. We shall construct the transition tree of $\mathbf{A}_L$, the automaton defined in Proposition 6.5.6, directly from L. It is then an easy matter to construct $\mathbf{A}_L$ which is the minimal automaton by Theorem 6.5.8.

(1) The root of the tree is L. For each $a \in A$ calculate $a^{-1}L$ using Proposition 6.5.3. Join L to $a^{-1}L$ by an arrow labelled a. Any repetitions should be closed with a $\times$.

(2) Subsequently, for each non-closed vertex M calculate $a^{-1}M$ for each $a \in A$ using Proposition 6.5.3. Close repetitions using $\times$.

(3) The algorithm terminates when all leaves are closed. Mark with double circles all labels containing ε. The tree is now the transition tree of $\mathbf{A}_L$, and so $\mathbf{A}_L$ can be constructed in the usual way.

□

Example 6.5.10 Let $A = \{a, b\}$ and $L = (a + b)^*aba(a + b)^*$. We find $\mathbf{A}_L$ using the algorithm above.

(1) $\varepsilon^{-1}L = L = L_0$. By Examples 6.5.1(5).

(2) $a^{-1}L_0 = L + baA^* = L_1$. By Examples 6.5.4(5).

(3) $b^{-1}L_0 = L = L_0$, closed. By Proposition 6.5.3(iii) and Examples 6.5.4(3), and Examples 6.5.4(2).

(4) $a^{-1}L_1 = L_1$, closed. By Proposition 6.5.3(ii) and Examples 6.5.4(3).

(5) $b^{-1}L_1 = L + aA^* = L_2$. By Proposition 6.5.3(ii) and Examples 6.5.4(3) (adapted).

(6) $a^{-1}L_2 = a^{-1}L + A^* = A^* = L_3$. By Proposition 6.5.3 and Examples 6.5.4(4).

(7) $b^{-1}L_2 = L = L_0$, closed. By Proposition 6.5.4 and Examples 6.5.4(3).

(8) $a^{-1}L_3 = A^* = L_3$, closed. By Examples 6.5.4(2).

(9) $b^{-1}L_3 = A^* = L_3$, closed. By Examples 6.5.4(2).

The states of $\mathbf{A}_L$ are therefore

$$\{L_0, L_1, L_2, L_3\},$$

with L_0 as the initial state. The only quotient of L that contains ε is L_3 and so this is the terminal state. The minimal automaton for L is therefore as follows:

$$\begin{tikzcd}
\arrow[r] & L_0 \arrow[loop above, "b"] \arrow[r, "a"] & L_1 \arrow[loop below, "a"] \arrow[r, "b"] & L_2 \arrow[r, "a"] \arrow[ll, bend left=60, "b"] & \boxed{L_3} \arrow[loop above, "{a,b}"]
\end{tikzcd}$$

□

We conclude this section by discussing the one drawback of the Method of Quotients. For the Method of Quotients to work, we have to recognise when two quotients are equal as in step (5) in Example 6.5.10 above. However, we saw in Section 5.1 that checking whether two regular expressions are equal is not always easy. If we do not recognise that two quotients are equal, then the machine we obtain will no longer be minimal. Here is another example.

Example 6.5.11 Consider the regular expression,

$$r = a^*(aa)^*.$$

We calculate $a^{-1}r = a^*(aa)^* + a(aa)^*$. This looks different from r. However,

$$a^*(aa)^* = (\varepsilon + a + a^2 + \ldots)(aa)^*,$$

and so $a(aa)^* \subseteq a^*(aa)^*$. It follows that $a^{-1}r = r$. □

Two questions are raised by this problem:

Question 1 Could Algorithm 6.5.9 fail to terminate?

Question 2 If it does terminate, what can we say about the automaton described by the transition tree?

The answer to Question 1 is 'yes' but, as long as we do even a small amount of checking, we can guarantee that the algorithm will always terminate. The answer to Question 2 is that if we fail to recognise when two quotients are the same, then we shall obtain an accessible deterministic automaton but not necessarily one that is reduced. It follows that once we have applied the Method of Quotients we should calculate the indistinguishability relation of the resulting automaton as a check.

Exercises 6.5

1. Prove Lemma 6.5.2.

2. Complete the proof of Proposition 6.5.6.

3. Let $A = \{a, b\}$. For each of the languages below find the minimal automaton using the Method of Quotients.

   (i) ab.

   (ii) $(a + b)^*a$.

   (iii) $(ab)^*$.

   (iv) $(ab + ba)^*$.

   (v) $(a + b)^*a^2(a + b)^*$.

   (vi) aa^*bb^*.

   (vii) $a(b^2 + ab)^*b^*$.

   (viii) $(a + b)^*aab(a + b)^*$.

4. Calculate the quotients of $\{a^nb^n \colon n \geq 0\}$.

6.6 Summary of Chapter 6

- *Reduction of an automaton*: From each deterministic automaton $\mathbf{A}$ we can construct an automaton $\mathbf{A}^r$ with the property that each pair of states in $\mathbf{A}$ is distinguishable and $L(\mathbf{A}^r) = L(\mathbf{A})$. The automata $\mathbf{A}^{ra}$ and $\mathbf{A}^{ar}$ are isomorphic and both are reduced and accessible.

- *Minimal automaton*: Each recognisable language L is recognised by an automaton that has the smallest number of states amongst all the automata recognising L: this is the minimal automaton for L. Such an automaton must be reduced and accessible, and any reduced and accessible automaton must be minimal for the language it recognises. Any two minimal automata for a language are isomorphic.

- *Method of Quotients*: The minimal automaton corresponding to the language described by a regular expression r can be constructed directly from r by calculating the quotients of r.

Solutions to exercises

S.1 Introduction to finite automata

S.1.1 Alphabets and strings

1. The set of prefixes is

   $$\varepsilon, \text{a, aa, aar, aard, aardv, aardva, aardvar, aardvark}$$

   The set of suffixes is

   $$\varepsilon, \text{k, rk, ark, vark, dvark, rdvark, ardvark, aardvark}$$

   The set of factors is as follows:

   length 0

   $$\varepsilon$$

   length 1

   a, r, d, v, k

   length 2

   aa, ar, rd, dv, va, rk

   length 3

   aar, ard, rdv, dva, var, ark

length 4

aard, ardv, rdva, dvar, vark

length 5

aardv, ardva, rdvar, dvark

length 6

aardva, ardvar, rdvark

length 7

aardvar, ardvark

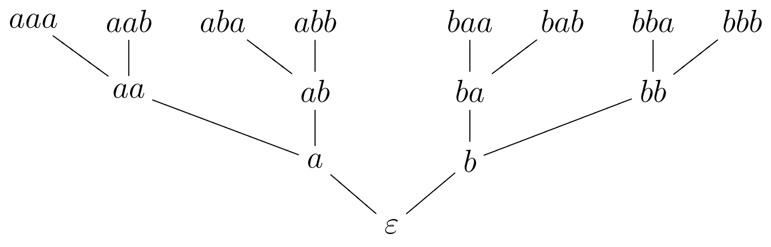
lenth 8

aardvark

Three substrings that are not factors

adk, rv, aardark

2. The tree is



The strings of length at most three arranged according to the tree order are

$$\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb$$

3. This simply follows from the definition of the equality of two strings: if $x = y$ then the first letter in x is the same as the first letter in y, the second letter in x is the same as the second letter in y, and so on. Similarly, the last letter in x is equal to the last letter in y, the second from last letter in x is equal to the second from last letter in y, and so on. More formally, from $xz = yz$ we get $|x| + |z| = |y| + |z|$ and so $|x| = |y|$. If $|x| = 0$ then $x = y = \varepsilon$. Otherwise $|x| = n > 0$. By assumption $(xz)_i = (yz)_i$ for all i and so in particular for those i such that $1 \leq i \leq n$. It follows that $x = y$, as required.

4. I shall prove (i) since the proof of (iii) is similar, and the proof of (ii) is immediate. We are given that $xy = uv$ and $|x| > |u|$. It follows that u is a prefix of x. Thus there is a string w such that $x = uw$. We may therefore write $xy = (uw)y$ and so $uwy = uv$. We cancel the u on both sides on the left to get $wy = v$. Hence $x = uw$ and $v = wy$.

5. We have to prove that (i)⇒(ii) and (ii)⇒(i). The proof of the second implication is easy, so I shall prove the first. We are given that $uv = vu$ and we have to prove that u and v are positive powers of one and the same string. Before giving the formal proof, let me explain how it was found; we take our cue from Question 4. Observe that if $|u| = |v|$ then $u = v$ and the result is immediate. Suppose that $|u| < |v|$. Then there is a string w such that $v = uw$ and $v = wu$. It follows that $uw = wu$ where $|u| + |w| < |u| + |v|$. Suppose that $u = z^p$ and $w = z^q$ for some string z and positive integers p and q. Then $u = z^p$ and $v = z^{p+q}$. This suggests that we should try to prove the result by induction on $n = |u| + |v|$. When $n = 2$ then the result is immediate since $ab = ba$ implies that $a = b$. This is our base case. Assume that for all n such that $2 \leq n \leq k$ we have that $uv = vu$, where $k = |u| + |v|$, implies that u and v are positive powers of one and the same string. This is our induction hypothesis. Let $n = k + 1$ and let $uv = vu$ be such that $k + 1 = |u| + |v|$. We have to prove that u and v are powers of one and the same string. We use Question 4 and the induction hypothesis. If $|u| = |v|$ then the result is immediate. If $|u| < |v|$ then our argument above can now be applied. The case $|v| < |u|$ follows from the above case since $uv = vu$ implies that $vu = uv$.

6. Let S be a semigroup with identities e and f. Then $ef = f$ since e is an identity but also $ef = e$ since f is an indenity. It follows that $e = f$,

as claimed.

7. (i) This is a semigroup because composition of functions is associative. (Can you prove this?)

   (ii) This is a semigroup because multiplication of matrices is associative. (Can you prove this?)

   (iii) This is not a semigroup: for example, $(\mathbf{i} \times \mathbf{i}) \times \mathbf{j} \neq \mathbf{i} \times (\mathbf{i} \times \mathbf{j})$.

S.1.2 Languages

I've not set any specific questions on this section, but I may set some in the homeworks.

S.1.3 Language operations

1. (i) $LM = \{ab, ba\}\{aa, ab\} = \{abaa, abab, baaa, baab\}$.

   (ii) $LN = \{ab, ba\}\{a, b\} = \{aba, abb, baa, bab\}$.

   (iii) $LM + LN = \{abaa, abab, baaa, baab, aba, abb, baa, bab\}$.

   (iv) $M + N = \{aa, ab\} + \{a, b\} = \{aa, ab, a, b\}$.

   (v) $L(M + N) = \{ab, ba\}\{aa, ab, a, b\}$ which is equal to

   $$\{abaa, abab, aba, abb, baaa, baab, baa, bab\}.$$

   (vi) $(LM)N = \{abaa, abab, baaa, baab\}\{a, b\}$ which is equal to

   $$\{abaaa, abaab, ababa, ababb, baaaa, baaab, baaba, baabb\}.$$

   (vii) $MN = \{aa, ab\}\{a, b\} = \{aaa, aab, aba, abb\}$.

   (viii) $L(MN) = \{ab, ba\}\{aaa, aab, aba, abb\}$ which is equal to

   $$\{abaaa, abaab, ababa, ababb, baaaa, baaab, baaba, baabb\}.$$

2. We have that

   $$a + b^* \subseteq a^* + b^* \subseteq (a^* + b^*)^*.$$

   The language $a + b^*$ consists of the letter a or arbitrary strings of b's. The language $a^* + b^*$ consists of arbitrary strings of a's or arbitrary strings of b's. The language $(a^* + b^*)^*$ consists of those strings that can be factorised into products of strings each of which is an arbitrary sequence of a's or an arbitrary sequence of b's. It follows that $(a^* + b^*)^* = (a + b)^*$.

3. All strings are over the alphabet $\{a, b\}$.

   (i) An arbitrary number of a's followed by an arbitrary number of b's.

   (ii) The empty string or strings that begin with a, end with b, and where a's and b's alternate.

   (iii) Strings of odd length.

   (iv) Strings that begin with a double letter.

   (v) Strings that contain at least one double letter as a factor.

   (vi) Strings that end with a double letter.

   (vii) Strings that contain as factors both aa and bb and where there is an occurrence of aa before an occurrence of bb.

4. If either x or y is the empty string then clearly $xy \in L^*$. We may therefore suppose that neither x nor y is the empty string. By definition, $x = x_1 \dots x_m$ for some $x_i \in L$ and $y = y_1 \dots y_n$ for some $y_j \in L$. It follows that xy can be written as a product of elements of L. Thus $xy \in L^*$, as required.

5. (i) We have to show that $(L^*)^* = L^*$. Clearly $L^* \subseteq (L^*)^*$. To prove the reverse inclusion, observe that an element of $(L^*)^*$ can be factorised as a product of elements of L^*. But a product of elements of L^* is again in L^*.

   (ii) We have to show that $L^*L^* = L^*$. The product of two elements in L^* is again in L^* so that $L^*L^* \subseteq L^*$. On the other hand, because $\varepsilon \in L^*$ we have that $L^* = L^*\varepsilon \subseteq L^*L^*$.

   (iii) We have to show that $L^*L + \varepsilon = L^* = LL^* + \varepsilon$. I shall prove that $L^*L + \varepsilon = L^*$, since the other case is similar. It is clear that $L^*L + \varepsilon \subseteq L^*$. To prove the reverse inclusion, observe that a non-empty element x of L^* can be written $x = uv$ where $u \in L^*$ and $u \in L$.

We are finally asked to determine if it is true that $LL^* = L^*$. By (iii) above, we have that $L^* = LL^* + \varepsilon$. Thus $LL^* = L^*$ iff $LL^* = LL^* + \varepsilon$ iff $\varepsilon \in LL^*$. But $\varepsilon \in L^*$ always, and so $LL^* = L^*$ iff $\varepsilon \in L$.

6. For (i) and (ii), use the general method of showing that two sets X and Y are equal: show that $X \subseteq Y$ and $Y \subseteq X$.

   (i) Let $x \in L(MN)$. Then $x = uv$ where $u \in L$ and $v \in MN$. But $v \in MN$ implies that $v = wz$ where $w \in M$ and $z \in N$. Thus $x = uv = u(wz) = (uw)z$ by associativity of concatenation. But $uw \in LM$ and so $x \in (LM)N$. It follows that we have proved that $L(MN) \subseteq (LM)N$. The reverse inclusion is proved similarly.

   (ii) I shall prove that $L(M + N) = LM + LN$, the proof of the other case is similar. Let $x \in L(M + N)$. Then $x = uv$ where $u \in L$ and $v \in M + N$. It follows that either $v \in M$ or $v \in N$. Thus $uv \in LM$ or $uv \in LN$. Hence $uv \in LM + LN$. We have proved that $L(M + N) \subseteq LM + LN$. The reverse inclusion is proved similarly.

   (iii) I shall prove that $NL \subseteq NM$, the proof of the other case is similar. Let $x \in NL$. Then $x = uv$ where $u \in N$ and $v \in L$. But $L \subseteq M$, and so $v \in M$. Hence $uv \in NM$ and so $x \in NM$, as required.

   (iv) I shall prove that $L\sum_{i=1}^{\infty} M_i = \sum_{i=1}^{\infty} LM_i$. Let $x \in L\sum_{i=1}^{\infty} M_i$. Then $x = lm$ where $l \in L$ and $m \in M_i$ for some i. It follows immediately that $x \in \sum_{i=1}^{\infty} LM_i$. Suppose now that $x \in \sum_{i=1}^{\infty} LM_i$. Then $x = lm$ where $l \in L$ and $m \in M_i$ for some i. Clearly $m \in \sum_{i=1}^{\infty} M_i$ and so $x \in L\sum_{i=1}^{\infty} M_i$.

7. It is easy to check that $L(M \cap N) \subseteq LM \cap LN$. To show that the reverse inclusion does not hold, let $L = \{\varepsilon, a\}$, $M = \{a, b\}$, and $N = \{aa, bb\}$. Then $L(M \cap N) = \emptyset$, whereas $LM \cap LN = \{aa\}$.

8. Prove first that the left-hand side is contained in the right-hand side. A string in $(ab)^+$ clearly begins with a and ends with b. In addition, it is clear that neither aa nor bb can occur as a factor. Thus the left-hand side is contained in the right-hand side. Now consider a string in the right-hand side. It must begin with an a and end with a b and neither aa nor bb can be a factor. The result is now clear.

9. The intersection $uA^* \cap vA^*$ is non-empty iff $ux = vy$ for some $x, y \in A^*$. Now $ux = vy$ implies that either u is a prefix of v or vice versa. Conversely, suppose that one of u and v is a prefix of the other. Without loss of generality, we suppose that u is a prefix of v. Then $v = uw$ for some string w. It follows that $ux = vy$ for some $x, y \in A^*$ iff u is a prefix of v or vice versa. Hence $uA^* \cap vA^*$ is non-empty iff u is a prefix of v or vice versa.

   Suppose that u is a prefix of v. Then $v = uw$ for some w. Thus $vA^* = uwA^* \subseteq uA^*$. Hence $uA^* \cap vA^* = vA^*$ if u is a prefix of v.

10. We have to determine when $L^+ = L^*$. Suppose $\varepsilon \in L$. Then

$$L^+ = L + L^2 + \ldots = \varepsilon + L + L^2 + \ldots = L^*.$$

   Conversely, if $L^+ = L^*$ then $\varepsilon \in L^+$ and so $\varepsilon \in L$. Hence $L^+ = L^*$ iff $\varepsilon \in L$.

11. In Question 4 above we proved that L^* is a submonoid containing L. Let T be any submonoid of A^* containing L. It is a submonoid so by assumption $\varepsilon \in T$. Since $L \subseteq T$ and T is closed under concatenation we must have that $L^2 \subseteq T$. But $L, L^2 \subseteq T$ implies that $L^3 = LL^2 \subseteq T$. In general, $L^n \subseteq T$. We have show that $\{\varepsilon\}, L, L^2, L^3, \ldots \subseteq T$. Thus the union of these languages is a subset of T. But this union is just L^*, and we have proved our claim.

12. $\mathsf{P}(A^*)$ is a monoid with respect to both $+$ and $\cdot$.

   If z and z' are both zeros then $zz' = z'$ and $zz' = z$ and so $z = z'$, as claimed.

   With respect to concatenation of languages $\mathsf{P}(A^*)$ is a monoid with zero where the zero is the empty set.

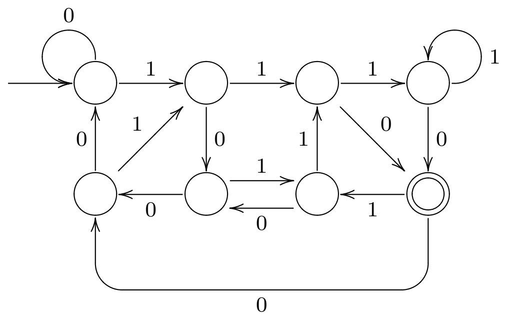
S.1.4 Finite automata: motivation

1. It is convenient to describe the automaton by means of a table: the left-hand column lists all the states; if we look at the row labelled by the state q then the entry in row q and column 0 tells us the next state, when the machine is in state q and 0 is input, likewise the entry in row q and column 1 tells us the next state, when the machine is in state q

and 1 is input. The state indicated by the arrow $\rightarrow$ is the initial state, and the state indicated by the arrow $\leftarrow$ is the terminal state.

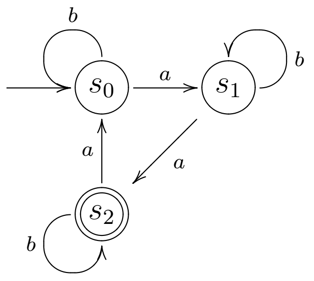
	0	1
000	000	001
$\leftarrow$ 001	010	011
$\rightarrow$ 010	100	101
100	000	001
011	110	111
101	010	011
110	100	101
111	110	111

The corresponding diagram is therefore as follows:

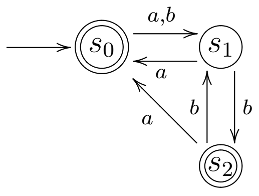


S.1.5 Finite automata and their languages

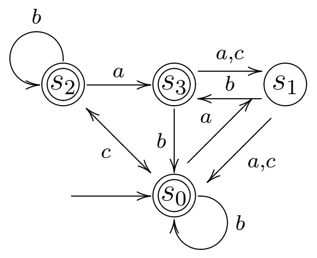
1. (i)



(ii)



(iii)



2. (i) This is ok — the set of terminal states can be empty.

   (ii) This is ok — the initial state can be terminal.

   (iii) This is ok — all states can be terminal.

   (iv) Not ok — two arrows emerge from the left-hand state with the same label a.

   (v) This is ok — an automaton does not have to be in one piece.

   (vi) Not ok — there is no transition emerging from the right-hand state labelled a.

3. This is the required table

	ε	a	b	a^2	ab	ba	b^2	a^3	a^2b	aba	ab^2	ba^2	bab	b^2a	b^3
1	1	2	1	2	3	2	1	2	3	4	1	2	3	2	1
2	2	2	3	2	3	4	1	2	3	4	1	4	4	2	1
3	3	4	1	4	4	2	1	4	4	4	4	2	3	2	1
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

4. (i) $a + a^3 + a^5(a^3)^*$.

   (ii) $a(a + b)^* + b(a + b)(a + b)^*$.

   (iii) $(ab)^*$.

   (iv) $b^*aa^*b(a + b)^*$.

5. If x is a binary string then $\#(x)$ is the (decimal) value of x. Observe that

$$\#(x0) = 2\#(x), \text{ and } \#(x1) = 2\#(x) + 1.$$

The idea is that the automaton we construct will have states $0, 1, 2$ which tells us whether the value of the string we have read so far is either divisible by 3 or leaves remainder 1 or 2. Thus for each possibility $\#(x) = 3n, 3n+1, 3n+2$ we calculate $\#(x0)$ and $\#(x1)$ as multiples of 3 plus a remainder.

6. (i) This is a problem in which the answer is either a 'yes' or a 'no'.
   (ii) The inputs are encoded over some finite alphabet. The encoded inputs that yield 'yes' form the language.
   (iii) A decision problem is decidable if there is an algorithm for deciding membership or not of the associated language.
   (iv) We encode simple graphs in the usual way. The binary strings encoding the problem are binary strings of length n^2 which consist entirely of 1's except in positions $1, n+1, 2n+2, \ldots, (n-1)n + (n-1)$. The membership of this language is clearly decidable.
   (v) Yes. The automaton recognising the language is itself an algorithm deciding membership.

S.2 Recognisable languages

S.2.1 Designing automata

1. It is easy to check that 1^+0 is accepted by $\mathbf{A}$. To show that not every string accepted by $\mathbf{A}$ is of this form, consider the string 101. It is clearly accepted by $\mathbf{A}$ but is not in 1^+0. The language accepted by $\mathbf{A}$ is 1^+01^*. To see why, observe that the bottom left-hand state once entered can never be escaped.

2. We have to show that every string in $L(\mathbf{A})$ has an odd number of 1's. There are two 'outward paths' that start at the initial state and finish at the terminal state: one starts at the initial state and uses the bottom left-hand state, and the other is a direct transition; in both cases, the symbol 1 occurs once. There are two 'return paths' that start at the terminal state and end at the initial state: one starts at the terminal state and uses the top right-hand state and the other is a direct transition; in both cases, the symbol 1 occurs once. Any path from the initial state to the terminal state must involve a path that is composed of a number of outward paths and one less the number of return paths. It follows that any successful path must contain an odd number of 1's.

However, the string 10 has an odd number of 1's but is not accepted. The language recognised by **A** is $(1+0^+1)((1+0^+1)(1+0^+1))^*$. To see why, observe that $1+0^+1$ labels outward paths and return paths.

3. Let L be the language consisting of an odd number of 1's. The string $1 \in L$ but $1 \notin L(\mathbf{A})$. The string $110 \in L(\mathbf{A})$ but $110 \notin L$. The language recognised by **A** is $((0+10^*1)(0+1))^*$. To see why, think about paths from the initial state to the bottom right-hand state, and paths from that state back to the initial state.

S.2.2 Automata over one letter alphabets

1. (i) $a+a^3+a^5(a^3)^*$.

   (ii) $\varepsilon+a^3+a^4+a^5(a^3)^*$.

   (iii) $a^3+a^5(a^3)^*$.

2. In each case, you have to 'normalise' the description of the language before you can construct an automaton.

   (i) $a^2+a^5+(a^2+a^3)(a^4)^* = a^2+a^5+a^2(a^4)^*+a^3(a^4)^*$. Now $a^2(a^4)^* = a^2+a^6+a^{10}+a^{14}+\ldots$ and $a^3(a^4)^* = a^3+a^7+a^{11}+\ldots$. If we add all terms together we get $a^2+a^3+a^5+(a^6+a^6(a^4)+\ldots)+(a^7+a^7(a^4)+\ldots)$ which is equal to $a^2+a^3+a^5+(a^6+a^7)(a^4)^*$. It is now easy to draw a fryingpan automaton recognising this language.

   (ii) The language is equal to $(a^2+a^4)(a^2)^*$.

   (iii) The language is equal to $a^2+a^4+a^5+(a^7+a^8)(a^3)^*$.

3. Let $L = X + Y(a^p)^*$. We shall prove that L is recognisable by constructing an automaton to accept L. This can be done easily if two conditions are met: first, if the length of the longest string in X is strictly less than the length of the shortest string in Y, and, second, if the length of the longest string in Y is strictly less than p plus the length of the shortest string in Y. If this is the case then the handle has length the shortest string in Y and the pan, of length p is attached to the last state in the handle. The terminal states in the handle are marked using the strings in X and the states in the pan marked using the strings in Y.

Consider, now, the case where these two conditions don't hold. For each string $y \in Y$ all the strings in $y(a^p)^* = y + y(a^p) + y(a^p)^2 + \ldots$ belong to the language. Choose an r so that all the strings in $Y(a^p)^r$ have length strictly greater than the length of the longest string in X. We have that

$$X + Y(a^p)^* = X + Y[(\varepsilon + a^p + \ldots + (a^p)^{r-1}) + (a^p)^r(a^p)^*]$$

which is equal to

$$X + Y(\varepsilon + a^p + \ldots + (a^p)^{r-1}) + Y(a^p)^r(a^p)^*.$$

Put

$$X' = X + Y(\varepsilon + a^p + \ldots + (a^p)^{r-1})$$

and

$$Y' = Y(a^p)^r.$$

Then

$$L = X' + Y'(a^p)^*$$

and every string in X' has length strictly less than every string in Y'. Next suppose that the length between the shortest string in Y' and the longest string in Y' is greater than p. Let $y \in Y'$ be the shortest string in Y'. All the strings $y(a^p)^*$ are in the language. Choose s such that $y(a^p)^s$ has length within p of the length of the longest string in Y'. Put the strings $y, y(a^p), \ldots, y(a^p)^{r-1}$ in the handle and replace y in Y' by $y(a^p)^s$. This process can be continued so that in the end the difference in length between the longest string in Y' and the shortest string in Y' is strictly less than p. We are then back to the case considered in the first paragraph and so the proof is complete.

4. Observe that $(a^3)^* + (a^4)^* = (\varepsilon + a^3 + a^4 + a^6 + a^8 + a^9)(a^{12})^*$. Thus $a^2((a^3)^* + (a^4)^*) = (a^2 + a^5 + a^8 + a^9 + a^{19} + a^{11})(a^{12})^*$. We can incorporate the language $a^3(a^3)^*$ by adding in a^3, a^6, a^9, a^{12}. To incorporate $a(a^4)^*$ we add in a, a^5. Thus the language is

$$(a + a^2 + a^3 + a^5 + a^6 + a^8 + a^9 + a^{10} + a^{11} + a^{12})(a^{12})^*.$$

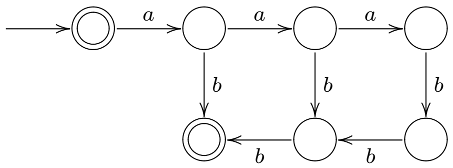
5. The longest string not in the language is a^7.

6. The length of the longest string not in the language is $pq - (p + q)$.

7. A language is 1-recognisable iff it is of the form $X + Y(a^p)^*$ where X and Y are finite sets and we can assume that the length of the shortest string in Y is longer than the length of the longest string in X. Let n be the length of the shortest string in Y. Then for all $m \geq n$ we have that a^m is in the language iff a^{m+p} is in the language, and so the corresponding subset of $\mathbb{N}$ is ultimately periodic. Conversely, suppose we have a set of numbers that is ultimately periodic. Then we can build a fryingpan automaton whose pan has size p and whose handle has length n. Mark as terminal states on the handle those states corresponding to the numbers strictly less than n in the subset. Mark as terminal states on the pan those numbers between (and including) n and strictly less than $n + p$ those numbers in the subset. The language recognised by this automaton corresponds to the set of numbers.

S.2.3 Incomplete automata

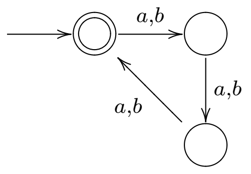
1. It is convenient to write down an incomplete automaton that does the job.



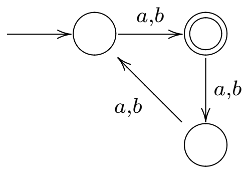
2. Let $L = \{x_1, \ldots, x_n\}$ be a finite language over the alphabet A. If $L = \emptyset$ then it is easy to construct an automaton recognising L, so we can assume that L is non-empty. Let m be the length of the longest string in L. Construct the tree for A^*, with labelled edges as in Example 2.3.3, up to and including those strings of length m. This tree can be converted into an incomplete automaton: mark ε as the initial state, and mark those vertices that belong to L as terminal. This incomplete automaton recognises L.

S.2.4 Automata that count

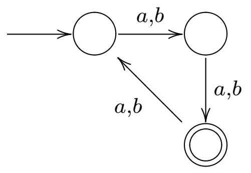
1. (i)



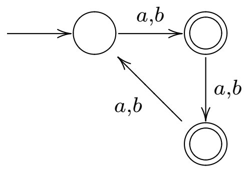
(ii)



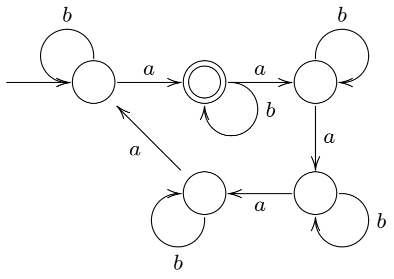
(iii)



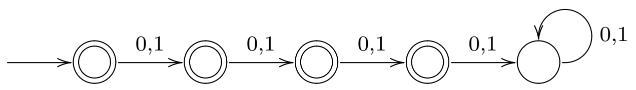
(iv)



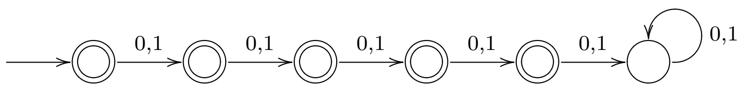
2.



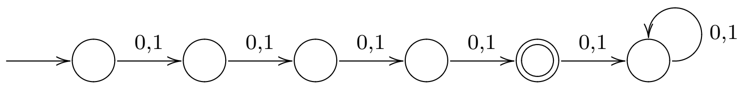
3. (i)



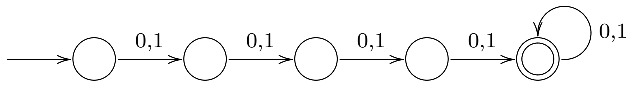
(ii)



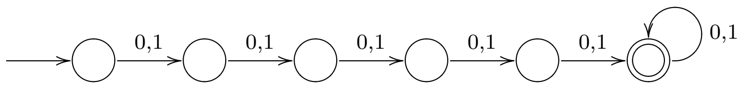
(iii)



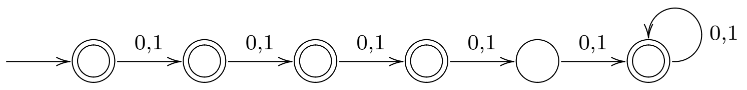
(iv)



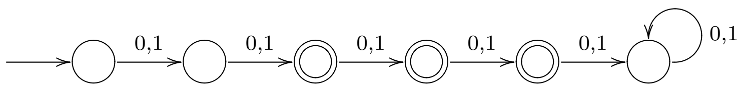
(v)



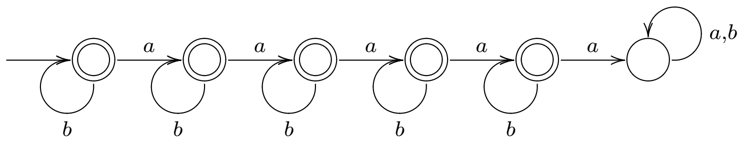
(vi)



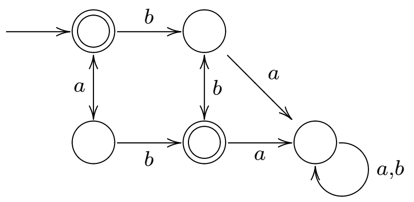
(vii)



4.

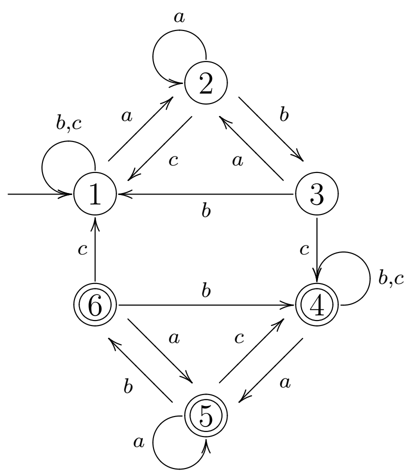


5.



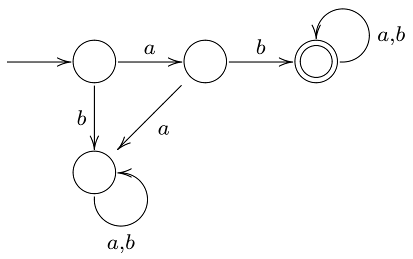
6.

	a	b	c
$\rightarrow 1$	2	1	1
2	2	3	1
3	2	1	4
$\leftarrow 4$	5	4	4
$\leftarrow 5$	5	6	4
$\leftarrow 6$	5	4	1

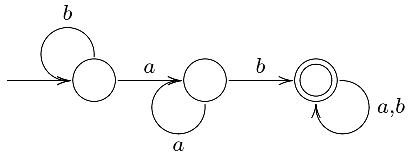


S.2.5 Automata that locate patterns

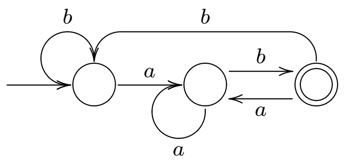
1. (i)



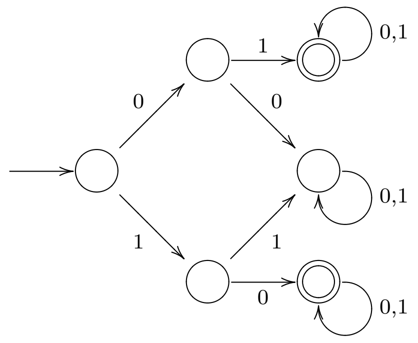
(ii)



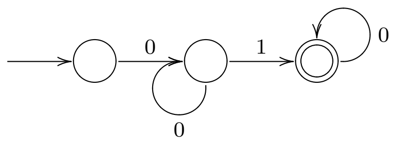
(iii)



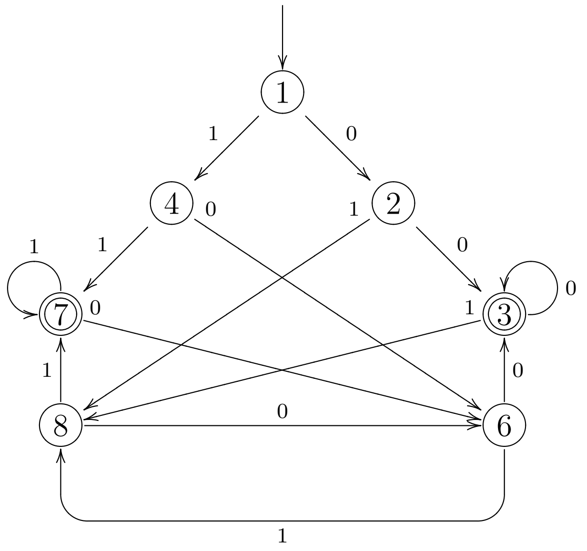
2. (i)



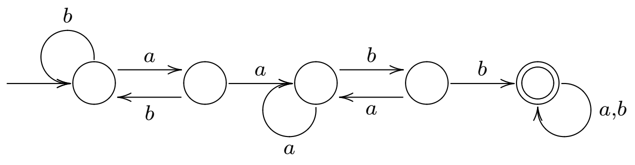
(ii) An incomplete machine that does the job is



(iii)



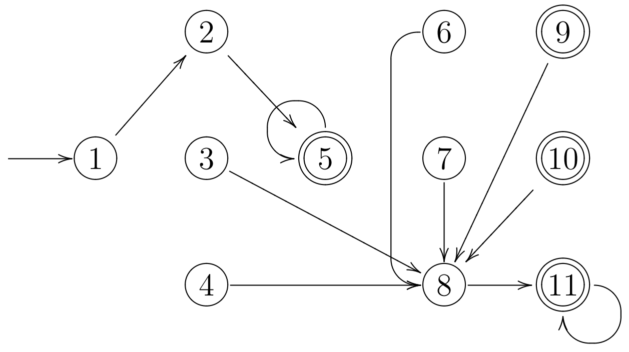
3.



4. The transition table of the machine is

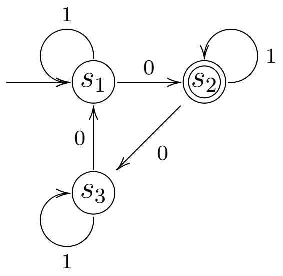
	a	b	c
$\rightarrow 1$	2	3	4
2	5	7	6
3	8	5	6
4	8	7	5
$\leftarrow 5$	5	5	5
6	8	7	9
7	8	10	6
8	11	7	6
$\leftarrow 9$	8	7	9
$\leftarrow 10$	8	10	6
$\leftarrow 11$	11	7	6

The diagram below shows the transitions labelled by a only, for the sake of clarity.

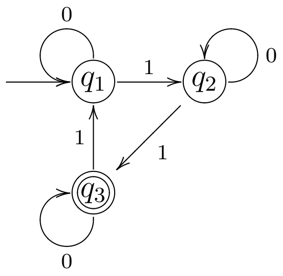


S.2.6 Boolean operations

1. An automaton **A** that recognises L is



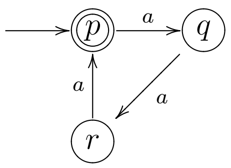
An automaton **B** that recognises M is



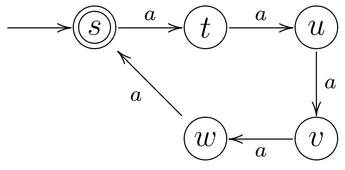
The transition table of the automaton $\mathbf{A} \times \mathbf{B}$ is therefore

	0	1
$\rightarrow (s_1, q_1)$	(s_2, q_1)	(s_1, q_2)
(s_2, q_1)	(s_3, q_1)	(s_2, q_2)
(s_3, q_1)	(s_1, q_1)	(s_3, q_2)
(s_1, q_2)	(s_2, q_2)	(s_1, q_3)
(s_2, q_2)	(s_3, q_2)	(s_2, q_3)
(s_3, q_2)	(s_1, q_2)	(s_3, q_3)
(s_1, q_3)	(s_2, q_3)	(s_1, q_1)
$\leftarrow (s_2, q_3)$	(s_3, q_3)	(s_2, q_1)
(s_3, q_3)	(s_1, q_3)	(s_3, q_1)

2. An automaton **A** that recognises L is



An automaton $\mathbf{B}$ that recognises M is



The transition table of the automaton $\mathbf{A} \sqcup \mathbf{B}$ is

	a
$\leftrightarrow (p,s)$	(q,t)
(q,t)	(r,u)
(r,u)	(p,v)
$\leftarrow (p,v)$	(q,w)
(q,w)	(r,s)
$\leftarrow (r,s)$	(p,t)
$\leftarrow (p,t)$	(q,u)
(q,u)	(r,v)
(r,v)	(p,w)
$\leftarrow (p,w)$	(q,s)
$\leftarrow (q,s)$	(r,t)
(r,t)	(p,u)
$\leftarrow (p,u)$	(q,v)
(q,v)	(r,w)
(r,w)	(p,s)

3. Observe that $L \setminus M = L \cap M'$. We are given that L and M are recognisable. Thus M' is recognisable by Proposition 2.6.2, and so $L \cap M'$ is recognisable by Proposition 2.6.5. Hence the result.

4. Let $L = L(\mathbf{A})$ and $M = L(\mathbf{B})$. Then $L + M = (L' \cap M')'$ by one of de Morgan's laws. Let $\mathbf{A}$ have set of states S and terminal states F, and let $\mathbf{B}$ have set of states T and terminal states G. Then $(\mathbf{A}' \times \mathbf{B}')'$ has set of terminal states $(S \times T) \setminus ((S \setminus F) \times (T \setminus G))$. But this set of terminal states is just $(F \times T) + (S \times G)$. It is now clear that $\mathbf{A} \sqcup \mathbf{B} = (\mathbf{A}' \times \mathbf{B}')'$.

5. Both results are proved by induction. I shall sketch out the proof for the case involving union. The case where $n = 2$ is the case proved by

Proposition 2.6.6. Suppose the result is true when $n = k$, we prove that the result is true when $n = k + 1$. Observe that we can write

$$L_1 + \ldots + L_{k+1} = (L_1 + \ldots + L_k) + L_{k+1}.$$

By the induction hypothesis, the language within the brackets, L say, is recognisable, and $L + L_{k+1}$ is recognisable by the base case. It follows that $L_1 + \ldots + L_{k+1}$ is recognisable.

6. Assume that L is recognisable. Then $L \cap a^*b^*$ is recognisable (why?). But this language is just the language of Proposition 2.4.4, which we know to be non-recognisable. Thus L cannot be recognisable.

S.3 Non-deterministic automata

S.3.1 Accessible automata

1. It is usually more convenient to draw the transition trees upside-down. All leaves are closed, so I omit the $\times$.

(i)

$$\begin{tikzcd}
& & p \arrow[dl, "0"'] \arrow[dr, "1"] & \\
& q \arrow[dl, "0"'] \arrow[d, "1"] & & q \\
p & q & &
\end{tikzcd}$$

(ii)

$$\begin{tikzcd}
& & p \arrow[dl, "a"'] \arrow[dr, "b"] & & \\
& q \arrow[dl, "a"'] \arrow[d, "b"] & & r \arrow[d, "a"'] \arrow[dr, "b"] & \\
p & s \arrow[dl, "a"'] \arrow[d, "b"] & & s & p \\
r & q & & &
\end{tikzcd}$$

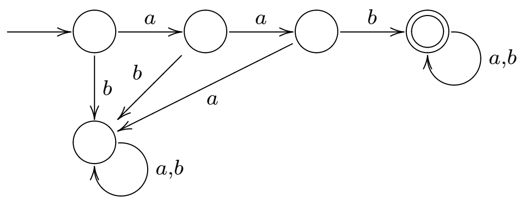
(iii)

$$\begin{tikzcd}
 & & & p \arrow[dl, "a"'] \arrow[dr, "b"] & & \\
 & & s \arrow[dl, "a"'] \arrow[d, "b"] & & u \arrow[d, "a"'] \arrow[dr, "b"] & \\
 & t \arrow[dl, "a"'] \arrow[d, "b"] & u & & u & u \\
t & t & & & &
\end{tikzcd}$$

S.3.2 Non-deterministic automata

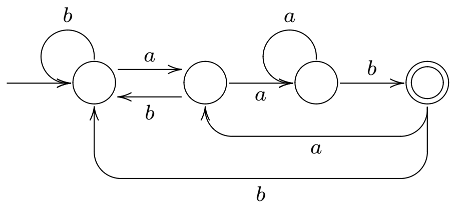
1. (i)

	a	b
$\to \{q\}$	$\{r\}$	$\emptyset$
$\{r\}$	$\{s\}$	$\emptyset$
$\{s\}$	$\emptyset$	$\{t\}$
$\leftarrow \{t\}$	$\{t\}$	$\{t\}$
$\emptyset$	$\emptyset$	$\emptyset$



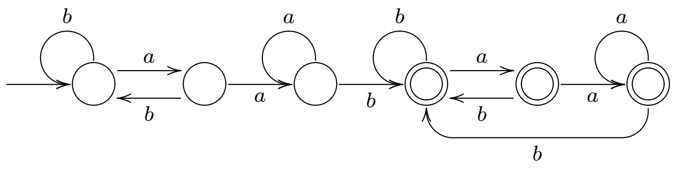
(ii)

	a	b
$\to \{q\}$	$\{q, r\}$	$\{q\}$
$\{q, r\}$	$\{q, r, s\}$	$\{q\}$
$\{q, r, s\}$	$\{q, r, s\}$	$\{q, t\}$
$\leftarrow \{q, t\}$	$\{q, r\}$	$\{q\}$



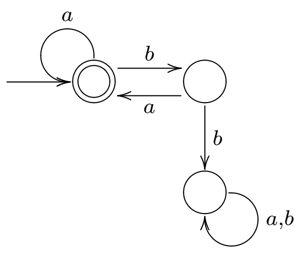
(iii)

	a	b
$\rightarrow \{q\}$	$\{q,r\}$	$\{q\}$
$\{q,r\}$	$\{q,r,s\}$	$\{q\}$
$\{q,r,s\}$	$\{q,r,s\}$	$\{q,t\}$
$\leftarrow \{q,t\}$	$\{q,r,t\}$	$\{q,t\}$
$\leftarrow \{q,r,t\}$	$\{q,r,s,t\}$	$\{q,t\}$
$\leftarrow \{q,r,s,t\}$	$\{q,r,s,t\}$	$\{q,t\}$

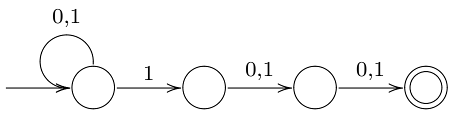


(iv)

	a	b
$\leftrightarrow \{s,t\}$	$\{s,t\}$	$\{s\}$
$\{s\}$	$\{s,t\}$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$



2. A non-deterministic automaton that recognises the language is



The transition table of the corresponding deterministic machine is

	0	1
$\rightarrow 0$	0	1
1	10	11
10	100	101
11	110	111
$\leftarrow 100$	0	1
$\leftarrow 101$	10	11
$\leftarrow 110$	100	101
$\leftarrow 111$	110	111

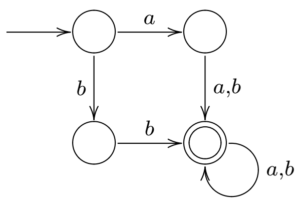
3. The non-deterministic automaton for this language is clear: it has the same basic shape as the non-deterministic automaton in Question 2, but has $n+1$ states.

   A deterministic automaton for this language must have enough states to 'remember' factors of the form $1x$ where x is a binary string of length at most $n-1$. The number of such strings is $2^0+2^1+\ldots+2^{n-1} = 2^n-1$. There must also be an initial state that remembers sequences of 0's not preceded by a 1, or factors of the form $(0^n)^+$. This gives us 2^n states.

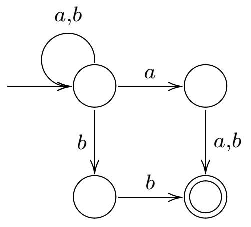
   A formal proof of this result will be given in answer to Exercises 6.4, Question 3.

S.3.3 Applications

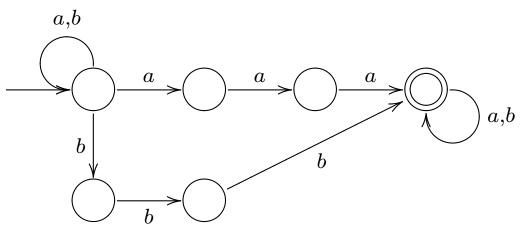
1. (i)



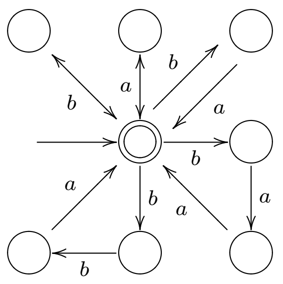
(ii)



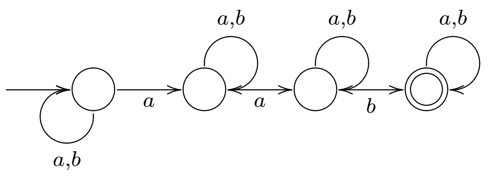
(iii)



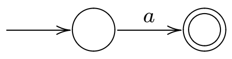
(iv)



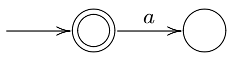
(v)



2. Consider the automaton



over the alphabet $A = \{a, b\}$. This recognises the language $\{a\}$. Now consider the automaton obtained from the above machine by making the current terminal state an ordinary state and all other states terminal:



This machine recognises the language $\{\varepsilon\}$.

S.4 ε-automata

S.4.1 Automata with ε-transitions

1. (i)

state $\star$	$E(\star)$	$E(\star)\cdot a$	$E(\star)\cdot b$	$E(E(\star)\cdot a)$	$E(E(\star)\cdot b)$
p	$\{p,q,t\}$	$\{r\}$	$\{u\}$	$\{r\}$	$\{u\}$
q	$\{q\}$	$\{r\}$	$\emptyset$	$\{r\}$	$\emptyset$
r	$\{r\}$	$\{s\}$	$\emptyset$	$\{s\}$	$\emptyset$
s	$\{s\}$	$\{s\}$	$\{s\}$	$\{s\}$	$\{s\}$
t	$\{t\}$	$\emptyset$	$\{u\}$	$\emptyset$	$\{u\}$
u	$\{u\}$	$\emptyset$	$\{v\}$	$\emptyset$	$\{v,s\}$
v	$\{v,s\}$	$\{s\}$	$\{s\}$	$\{s\}$	$\{s\}$

In this case, ε is not accepted and so there is no need for an extra state. The automaton $\mathbf{A}^{sda}$ is

$$\begin{tikzcd}
{} \arrow[r] & p \arrow[r, "a"] \arrow[d, "b"'] & r \arrow[d, "b"] \arrow[dd, bend left, "a"] \\
 & u \arrow[r, "a"] \arrow[d, "b"'] & \emptyset \arrow[loop below, "{a,b}"] \\
 & {v,s} \arrow[r, "{a,b}"] & s \arrow[loop below, "{a,b}"]
\end{tikzcd}$$

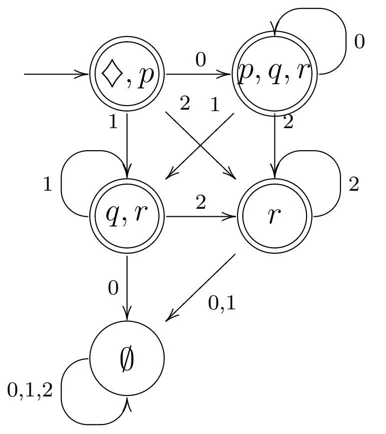
The language $L(\mathbf{A}) = (a^2 + b^2)(a + b)^*$.

(ii)

state $\star$	$E(\star)$	$E(\star)\cdot 0$	$E(\star)\cdot 1$	$E(\star)\cdot 2$
p	$\{p,q,r\}$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{q,r\}$	$\emptyset$	$\{q\}$	$\{r\}$
r	$\{r\}$	$\emptyset$	$\emptyset$	$\{r\}$

state $\star$	$E(E(\star)\cdot 0)$	$E(E(\star)\cdot 1)$	$E(E(\star)\cdot 2)$
p	$\{p,q,r\}$	$\{q,r\}$	$\{r\}$
q	$\emptyset$	$\{q,r\}$	$\{r\}$
r	$\emptyset$	$\emptyset$	$\{r\}$

In this case, ε is accepted and there is a need for an extra state. The automaton $\mathbf{A}^{sda}$ is

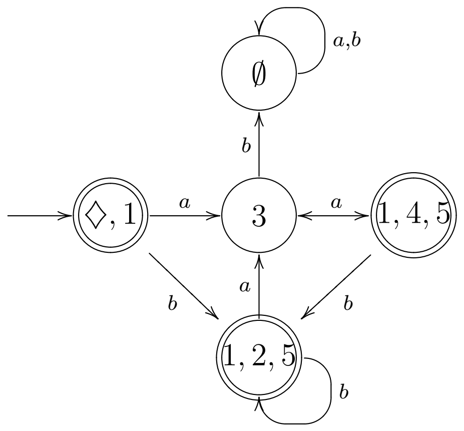


The language $L(\mathbf{A}) = 0^*1^*2^*$.

(iii)

state $\star$	$E(\star)$	$E(\star)\cdot a$	$E(\star)\cdot b$	$E(E(\star)\cdot a)$	$E(E(\star)\cdot b)$
1	$\{1,5\}$	$\{3\}$	$\{2\}$	$\{3\}$	$\{1,2,5\}$
2	$\{1,2,5\}$	$\{3\}$	$\{2\}$	$\{3\}$	$\{1,2,5\}$
3	$\{3\}$	$\{4\}$	$\emptyset$	$\{1,4,5\}$	$\emptyset$
4	$\{1,4,5\}$	$\{3\}$	$\{2\}$	$\{3\}$	$\{1,2,5\}$
5	$\{1,5\}$	$\{3\}$	$\{2\}$	$\{3\}$	$\{1,2,5\}$

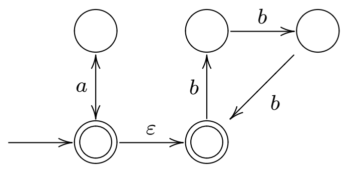
In this case, ε is accepted and there is a need for an extra state. The automaton $\mathbf{A}^{sda}$ is



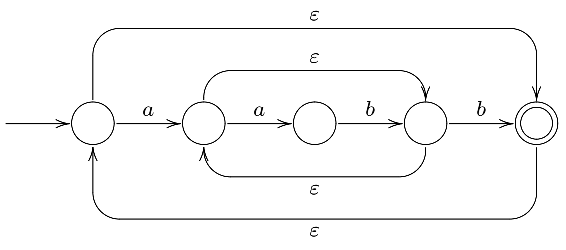
The language $L(\mathbf{A}) = (b + a^2)^*$.

S.4.2 Applications of ε-automata

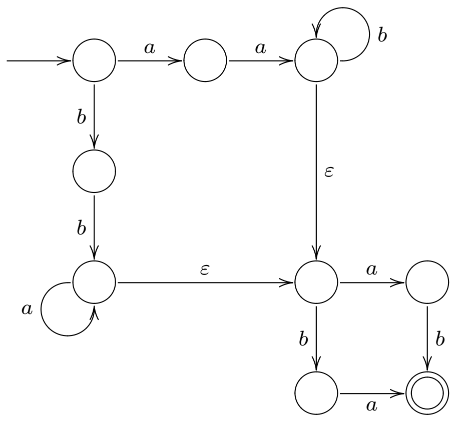
1. (i)



(ii)



(iii)



S.5 Kleene's Theorem

S.5.1 Regular languages

1. (i) $(a^3 + b)^*$.

   (ii) $b^*ab^*ab^*ab^*$.

   (iii) $b^*ab^*ab^* + b^*ab^*ab^*ab^*$.

   (iv) $(b + ab + a^2b)^*(\varepsilon + a + a^2)$.

   (v) $b^* + (b^*ab^*ab^*ab^*)^*$.

(vi) $(a+b)^*(a^2+b^2)$.

(vii) This is a little trickier. Let's consider first the case where the double letter is aa. The strings we want are of the form $x(aa)y$ where x has no double letters and does not end in a, and y has no double letters and does not begin with a. A string with no double letters must have letters that alternate. Thus they are of the form $(\varepsilon+b)(ab)^*(\varepsilon+a)$. Thus the set of strings where the only double letter is aa is described by

$$(\varepsilon+b)(ab)^*aa[b(ab)^*(\varepsilon+a)+\varepsilon].$$

But $b(ab)^* = (ba)^*b$. Thus we get, with a little calculation,

$$(\varepsilon+b)(ab)^*aa(ba)^*(\varepsilon+b).$$

Hence the answer to the question is

$$(\varepsilon+b)(ab)^*aa(ba)^*(\varepsilon+b)+(\varepsilon+a)(ba)^*bb(ab)^*(\varepsilon+a).$$

2. (i) By definition $r^* = \varepsilon + r + r^2 + r^3 + \ldots$. Rearrange the right-hand side into sums of even and odd powers:

$$r^* = (\varepsilon + r^2 + r^4 + \ldots) + (r + r^3 + r^5 + \ldots).$$

The first term is just $(rr)^*$ and the second term is $r(rr)^*$. Hence the result.

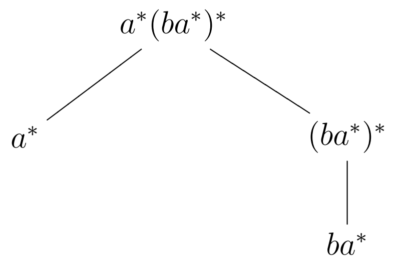
(ii) We need only prove that $(r+s)^* \subseteq (r^*s^*)^*$ (or, rather, the languages described by these regular expressions.) The left-hand side is the sum of terms of the form $(r+s)^n$, and $(r+s)^n$ is the sum of terms of the form $x^{n_1}\ldots x^{n_s}$ where $n_1+\ldots+n_s = n$ and each x_i is r or s. But such a term is of the form $(r^*s^*)^t$ for some t. Hence result.

(iii) By definition $(rs)^*r$ is the sum of terms of the form $(rs)^n r$ which is equal to the sum of terms of the form $r(sr)^n$ which is equal to $r(sr)^*$.

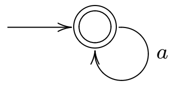
3. These are just a sequence of easy verifications. I shall prove (iv), as an example. Let $x \in L\cdot(M\cdot N)$. Then $x = ly$ where $l \in L$ and $y \in M\cdot N$. But $y \in M\cdot N$ implies that $y = mn$ where $m \in M$ and $n \in N$. Thus $x = l(mn) = (lm)n \in (L\cdot M)\cdot N$. Hence $L\cdot(M\cdot N) \subseteq (L\cdot M)\cdot N$. The reverse inclusion is proved similarly.

S.5.2 An algorithmic proof of Kleene's theorem

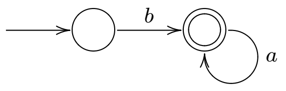
1. (i) The tree for this regular expression is



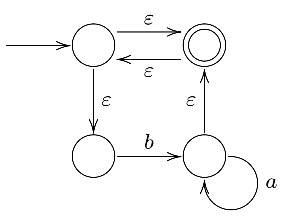
An automaton can now be constructed in steps from the tree. Automata for the leaves of this tree are



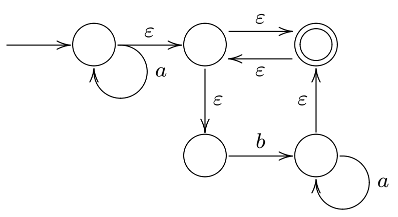
and



The automaton for $(ba^*)^*$ is



Finally, we can construct the automaton for $a^*(ba^*)^*$



(ii) The tree for this regular expression is

$$\begin{tikzcd}[arrows=-]
 & (a^*b+b^+a)^* \arrow[d] & \\
 & a^*b+b^+a \arrow[dl] \arrow[dr] & \\
a^*b & & b^+a
\end{tikzcd}$$

An automaton can now be constructed in steps from the tree.

(iii)

$$\begin{tikzcd}[arrows=-]
 & (a^2+b)^*(a+b^2)^* \arrow[dl] \arrow[dr] & \\
(a^2+b)^* \arrow[d] & & (a+b^2)^* \arrow[d] \\
a^2+b & & a+b^2
\end{tikzcd}$$

An automaton can now be constructed in steps from the tree.

2. (i) $b^*aa^*b(a+b)^*$.

   (ii) $a^*b(ab^*a^*b)^*$.

   (iii) $(a+ba+b^2)(a+b)^*$.

   (iv) $(ab)^*$.

   (v) $a+a^3+a^5(a^3)^*$.

3. We have to prove Lemma 5.2.3. Let $\mathbf{A}$ be a normalised automaton, and let $x \in L(\mathbf{A})$. Then by definition, we can factorise $x = x_1 \dots x_n$ in such a way that

$$\begin{tikzcd}
{} \arrow[r] & \alpha \arrow[r, "r_1"] & q_1 \arrow[r, "r_2"] & q_2 \arrow[r, dashed, no head] & q_{n-1} \arrow[r, "r_n"] & \omega
\end{tikzcd}$$

is a path in $\mathbf{A}$ where $x_i \in L(r_i)$.

Let $\mathbf{B}$ be the normalised automaton that results when one of the operations (T), (L) or (S) is applied. If none of the states in the path is affected by one of these rules, then clearly $x \in L(\mathbf{B})$. So we shall assume that some states on this path are affected by one of the rules. Suppose first, that rule (T) has been applied. Then it is immediate that $x \in L(\mathbf{B})$. Next, suppose that rule (L) has been applied to a loop on state q with label r. Then q occurs somewhere in our path in a sequence of consecutive positions. Thus $x = uy_1 \dots y_r v$, where $y_j \in L(r)$, and u labels a path from α to q, and v labels a path that starts at q and ends at ω. But $y_1 \dots y_r \in L(r^*)$, and so $x \in L(\mathbf{B})$. Finally, suppose that rule (S) has been applied with the elimination of state q. Then q occurs somewhere in our path. Thus $x = uyzv$ where $y \in L(r)$, $z \in L(s)$, and where y labels a transition that ends at q and s labels a transition that starts at q. Now $yz \in L(rs)$, and so $x \in L(\mathbf{B})$. Hence $L(\mathbf{A}) \subseteq L(\mathbf{B})$. The reverse inclusion is proved in a similar way, but working backwards in each case.

4. (i) Simply substitute C^*R into $CX + R$ and check that you get C^*R.

   (ii) Since $Y = CY + R$ we have that $R \subseteq Y$. We have that $C^2Y + CR + R = Y$ and so $CR \subseteq Y$. Continuing in this way we get that $CR^n \subseteq Y$ for all $n \geq 0$. It follows that $C^*R \subseteq Y$.

   (iii) Observe that $z \notin R$ but $z \in CW + R$. It follows that $z \in CW$. Hence $z = cw$ where $c \in C$ and $w \in W$. We now invoke our assumption: $c \neq \varepsilon$. It follows that $|w| < |z|$. Suppose that $w \in C^*R$. Then $z \in C^*R$. Contradiction. Thus $z \in W \setminus C^*R$. Contradiction.

S.6 Minimal automata

S.6.1 Partitions and equivalence relations

1. There are 15 equivalence relations on the set $\{1, 2, 3, 4\}$.

   (i) The partitions are:

$$\{\{1, 2, 3, 4\}\},$$

$$\{\{1\}, \{2, 3, 4\}\}, \{\{2\}, \{1, 3, 4\}\}, \{\{3\}, \{1, 2, 4\}\}, \{\{4\}, \{1, 2, 3\}\},$$

$\{\{1,2\},\{3,4\}\}$, $\{\{1,3\},\{2,4\}\}$, $\{\{1,4\},\{2,3\}\}$,

$\{\{1\},\{2\},\{3,4\}\}$, $\{\{3\},\{4\},\{1,2\}\}$, $\{\{2\},\{3\},\{1,4\}\}$,
$\{\{1\},\{4\},\{2,3\}\}$, $\{\{2\},\{4\},\{1,3\}\}$, $\{\{1\},\{3\},\{2,4\}\}$,

$\{\{1\},\{2\},\{3\},\{4\}\}$.

(ii) I shall give one example of a set of ordered pairs. Consider the partition $\{\{1,4\},\{2,3\}\}$. Then the set of ordered pairs that corresponds to this partition is

$$\{(1,1),(2,2),(3,3),(4,4),(1,4),(4,1),(2,3),(3,2)\}.$$

(iii) I shall give one example of the table form. The table form of the partition $\{\{1,4\},\{2,3\}\}$ is

	1	2	3	4
1	$\surd$	$\times$	$\times$	$\surd$
2	$*$	$\surd$	$\surd$	$\times$
3	$*$	$*$	$\surd$	$\times$
4	$*$	$*$	$*$	$\surd$

2. Let P be a partition on the set X. Define

$$x \sim_P y \Leftrightarrow x \text{ and } y \text{ belong to the same block of } P.$$

We prove that $\sim_P$ is reflexive, symmetric, and transitive. First, let $x \in X$. Then by (P3), x belongs to some block B, say. Clearly, x belongs to the same block as x. Hence $x \sim_P x$, and so $\sim_P$ is reflexive. Suppose that $x \sim_P y$. Then x and y belong to the same block of P. It follows that y and x belong to the same block of P. Hence $y \sim_P y$, and so $\sim_P$ is symmetric. Finally, suppose that $x \sim_P y$ and $y \sim_P z$. Then x and y belong to the same block B of P, and y and z belong to the same block B' of P. By (P2), we must have that $B = B'$ and so x and z belong to the same block of P. It follows that $x \sim_P z$, and so $\sim_P$ is transitive.

$\rho(y) \subseteq \sigma(x)$.

S.6.2 The indistinguishability relation

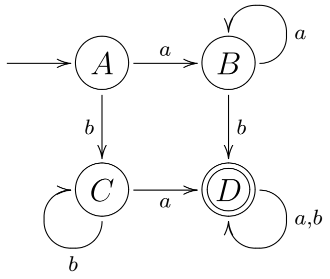
1. We prove that $\simeq_{\mathbf{A}}$ is an equivalence relation. Let $s \in S$ be a state. Then $s \cdot x \in T \Leftrightarrow s \cdot x \in T$ for all $x \in A^*$. Thus $s \simeq_{\mathbf{A}} s$. Suppose that $s \simeq_{\mathbf{A}} t$. Then $s \cdot x \in T \Leftrightarrow t \cdot x \in T$ for all $x \in A^*$. Hence $t \cdot x \in T \Leftrightarrow s \cdot x \in T$ for all $x \in A^*$. It follows that $t \simeq_{\mathbf{A}} s$. Finally, suppose that $s \simeq_{\mathbf{A}} t$ and $t \simeq_{\mathbf{A}} u$. Suppose that $s \cdot x \in T$. Then $t \cdot x \in T$ and so $u \cdot x \in T$. Thus $s \cdot x \in T \Rightarrow u \cdot x \in T$ for all $x \in A^*$. It is straightforward to prove the converse. Thus $s \simeq_{\mathbf{A}} u$.

2. In Theorem 6.2.3, we proved that $[s] \cdot a = [s \cdot a]$ for each $a \in A$ is well-defined. Clearly, $[s] \cdot \varepsilon = [s] = [s \cdot \varepsilon]$. Suppose that $[s] \cdot x = [s \cdot x]$ for all strings x of length n and all states s. Let y be a string of length $n+1$. Then we can write $y = ax$ where $a \in A$ and x has length n. By the definition of the extended transition function $[s] \cdot y = [s] \cdot (ax) = ([s] \cdot a) \cdot x$. By definition $([s] \cdot a) \cdot x = [s \cdot a] \cdot x$. By the induction hypothesis $[s \cdot a] \cdot x = [(s \cdot a) \cdot x] = [s \cdot (ax)] = [s \cdot y]$, as required.

3. (i) The indistinguishability relation is

$$\{A = \{1\}, B = \{2\}, C = \{4\}, D = \{3, 5, 6, 7\}\}.$$

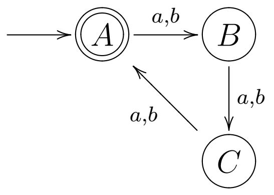
The machine $\mathbf{A}^r$ is



(ii) The indistinguishability relation is

$$\{A = \{0, 5\}, B = \{1, 2\}, C = \{3, 4\}\}.$$

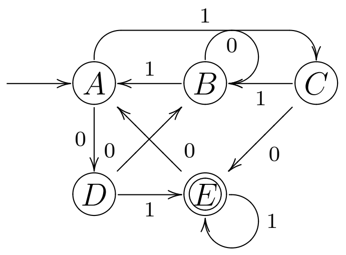
The machine $\mathbf{A}^r$ is



(iii) The indistinguishability relation is

$$\{A = \{1,5\}, B = \{7\}, C = \{4,6\}, D = \{2,8\}, E = \{3\}\}.$$

The machine $\mathbf{A}^r$ is



4. Let $\mathbf{A} = (S, A, i, \delta, \{t\})$ be an automaton with a unique terminal state such that for each $s \in S$ there exists $x \in A^*$ with $s \cdot x = t$, and such that τ_a is a bijection for each $a \in A$. We prove that $\mathbf{A}$ is reduced. Observe first that we may define τ_x for each $x \in A^*$ by τ_x maps s to $s \cdot x$. The function τ_ε is the identity function, and for each $x = a_1 \ldots a_n$, where $a_i \in A$, we have that $\tau_x = \tau_{a_1} \ldots \tau_{a_n}$, the composite of the functions τ_{a_i}. Since the composite of bijections is a bijection, it follows that τ_x is a bijection for all $x \in A^*$. Suppose that $s \simeq_{\mathbf{A}} s'$ in $\mathbf{A}$. Then $s \cdot x \in \{t\} \Leftrightarrow s' \cdot x \in \{t\}$ for all $x \in A^*$. We shall prove that $s = s'$. By assumption, there is a string x such that $s \cdot x = t$. Thus $s' \cdot x = t$. It follows that τ_x maps both s and s' to the same element. But τ_x is a bijection, and so $s = s'$, as required.

S.6.3 Isomorphisms of automata

1. (i) Let $\mathbf{A} = (S, A, s_0, \delta, F)$. The identity function 1_S: $S \to S$ satisfies (IM1)–(IM4). Thus $\mathbf{A} \equiv \mathbf{A}$.

(ii) Let $\mathbf{A} = (S, A, s_0, \delta, F)$ and $\mathbf{B} = (Q, A, q_0, \gamma, G)$. Let θ: $\mathbf{A} \to \mathbf{B}$. We prove that θ^{-1} is an isomorphism from $\mathbf{B}$ to $\mathbf{A}$. It is easy to check that (IM1)–(IM3) hold. We prove (IM4). Let $q \in Q$ and $a \in A$. Let $s \in S$ be such that $\theta(s) = q$. By assumption $\theta(s \cdot a) = \theta(s) \cdot a = q \cdot a$. Thus $\theta^{-1}(q \cdot a) = s \cdot a = \theta^{-1}(q) \cdot a$, as required.

(iii) Let θ: $\mathbf{A} \to \mathbf{B}$ and ϕ: $\mathbf{B} \to \mathbf{C}$ be isomorphisms. It is easy to check that $\phi\theta$ is an isomorphism from $\mathbf{A}$ to $\mathbf{C}$.

2. (i) An isomorphism is, in particular, a bijective function between the sets of states. Thus the sets of states must have the same number of elements.

   (ii) By (IM3), the bijection between the sets of states gives a bijection between the sets of terminal states.

   (iii) Suppose that $\mathbf{A}$ is accessible. We prove that $\mathbf{B}$ is accessible. Let $q \in Q$ be a state in $\mathbf{B}$. Then $s = \theta^{-1}(q)$ is a state in $\mathbf{A}$. By assumption, $s_0 \cdot x = s$ for some $x \in A^*$. By Lemma 7.3.1, we have that $\theta(s_0 \cdot x) = \theta(s_0) \cdot x$. By (IM2), we have that $\theta(s_0) = q_0$. Thus $q = \theta(s) = q_0 \cdot x$, and so q is accessible. Hence $\mathbf{B}$ is accessible.

   (iv) Suppose that $\mathbf{A}$ is reduced. We prove that $\mathbf{B}$ is reduced. Let

$$q \simeq_{\mathbf{B}} q'.$$

   Let $\theta(s) = q$ and $\theta(s') = q'$. We claim that $s \simeq_{\mathbf{A}} s'$. To see why, suppose that $s \cdot x \in F$. Then $\theta(s \cdot x) \in G$ by (IM3). But $\theta(s \cdot x) = \theta(s) \cdot x = q \cdot x$, by Lemma 6.3.1. Thus $q \cdot x \in G$. Hence $q' \cdot x \in G$. It follows that $\theta^{-1}(q' \cdot x) \in F$, by (IM3). But θ^{-1} is an isomorphism, and so $\theta^{-1}(q' \cdot x) = \theta^{-1}(q') \cdot x = s' \cdot x$. Hence $s' \cdot x \in F$. Thus $s \cdot x \in F$ implies $s' \cdot x \in F$. The converse is proved similarly. Hence $s \simeq_{\mathbf{A}} s'$. But $\mathbf{A}$ is reduced, and so $s = s'$. Hence $q = q'$. It follows that $\mathbf{B}$ is reduced.

3. Let the set of states of $\mathbf{A}$ be S with initial state s_0, and let the set of states of $\mathbf{B}$ be Q with initial state q_0. Let s be an arbitrary state in $\mathbf{A}$. Then $s = s_0 \cdot x$ for some $x \in A^*$ because $\mathbf{A}$ is accessible. Then $\theta(s) = \theta(s_0) \cdot x = q_0 \cdot x$ and $\phi(s) = \phi(s_0) \cdot x = q_0 \cdot x$. It follows that $\theta(s) = \phi(s)$. Since s was arbitrary, we have that $\theta = \phi$.

S.6.4 The minimal automaton

1.

subset	rank
$\emptyset$	1
00	4
01	4
10	4
11	4
$00 + 01$	4
$00 + 10$	4
$00 + 11$	5
$01 + 10$	5
$01 + 11$	4
$10 + 11$	4
$00 + 01 + 10$	5
$00 + 01 + 11$	5
$00 + 10 + 11$	5
$01 + 10 + 11$	5
$00 + 01 + 10 + 11$	3

2. A machine **A** that recognises this language consists of n states $s_0, \dots, s_{n-1}$ arranged in a circle with the initial state s_0 being terminal and where $s_i \cdot a = s_{i+1}$, $s_i \cdot b = s_{i+1}$ for $i = 0, \dots, n-2$ and $s_{n-1} \cdot a = s_0$, and $s_{n-1} \cdot b = s_0$. Observe that **A** is accessible and satisfies the conditions of Question 4 of Exercises 6.2. It follows that **A** is also reduced. Since **A** is an accessible reduced automaton recognising the language, it is a minimal automaton for the language.

3. We shall prove that a minimal automaton for the language has 2^n states. We construct a machine whose states are labelled by the 2^n strings

$$\varepsilon + \sum_{i=0}^{n-1} 1(0+1)^i.$$

The inital state is labelled ε the terminal states are those labelled by the strings of length n. For all states which are not terminal or initial input letter a takes state x to state xa. The letter 0 labels at loop at the initial state, and the letter 1 takes the initial state to the one labelled 1. If x labels a terminal state and a is an input letter then we map to the state given by the longest suffix of xa that begins with a 1 and whose length is at most n. I leave it as an exercise to prove that this machine recognises the given language and is accessible. The result will therefore be proved if we can show that this machine is reduced.

It is easy to see that states labelled by strings of different lengths are distinguishable. If we can show that all the terminal states are distinguishable it will follow that all the other pairs of states having the same sized label are distinguishable from the tree structure of the states. It remains to prove that any two terminal states are distinguishable. Let x and y label two terminal states. Reading from left to right suppose they disagree for the first time at the ith position. We can suppose without loss of generality that x has a 0 there and y a 1. Let u be any string of length i. Then $x \cdot u$ will be a non-terminal state and $y \cdot u$ will be a terminal state.

4. (i) This is straightforward.
   (ii) We use (i). Let $x \in L(\mathbf{A})$. Then $s_0 \cdot x \in F$. Thus $\theta(s_0 \cdot x) = q_0 \cdot x \in G$. Hence $x \in L(\mathbf{B})$.
   (iii) Let θ and ϕ both be homomorphisms from $\mathbf{A}$. By definition they agree on s_0. Let s be an arbitrary state of $\mathbf{A}$. By assumption $s = s_0 \cdot x$. It follows quickly that $\theta(s) = \phi(s)$ and so $\theta = \phi$.
   (iv) Straightforward.
   (v) Define θ by $\theta(s_0) = q_0$ and if $s = s_0 \cdot x$ then $\theta(s) = q_0 \cdot x$. The fact that $\mathbf{B}$ is reduced shows that θ is well-defined. The proof of the rest is straightforward.

S.6.5 The method of quotients

1. The proofs are very simple. For example, it is clear that $\varepsilon \in LM$ iff $\varepsilon \in L$ and $\varepsilon \in M$. Thus $\delta(LM) = \delta(L) \cap \delta(M)$.

2. To complete the proof of Proposition 6.5.6, we have to show that

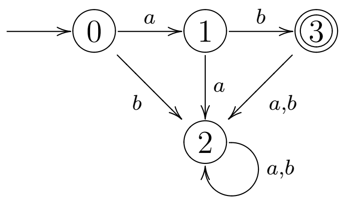
$$\delta^*(u^{-1}L, x) = (ux)^{-1}L$$

for all $x \in A^*$ and $u \in A^*$. The result is clearly true when $x = \varepsilon$ and when $x = a \in A$. Assume the result is true for all strings x of length

n. Let y be a string of length $n+1$. Then $y = ax$ where $a \in A$ and x has length n. Then $\delta^*(u^{-1}L, y) = \delta^*(u^{-1}L, ax) = \delta^*(\delta(u^{-1}L, a), x)$. This equals $\delta^*((ua)^{-1}L, x)$ using the base case, and this in turn equals $((ua)x)^{-1}L = (uy)^{-1}L$, as required.

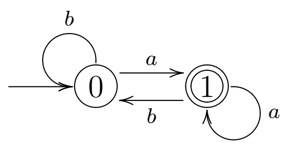
3. (i)
   - $\varepsilon^{-1}L = L$.
   - $a^{-1}L = b = L_1$.
   - $b^{-1}L = \emptyset = L_2$.
   - $a^{-1}L_1 = L_2$.
   - $b^{-1}L_1 = \varepsilon = L_3$.
   - $a^{-1}L_2 = L_2$.
   - $b^{-1}L_2 = L_2$.
   - $a^{-1}L_3 = L_2$.
   - $b^{-1}L_3 = L_2$.

   The minimal automaton is therefore



   (ii)
   - $\varepsilon^{-1}L = L$.
   - $a^{-1}L = L + \varepsilon = L_1$.
   - $b^{-1}L = L$.
   - $a^{-1}L_1 = L_1$.
   - $b^{-1}L_1 = L$.

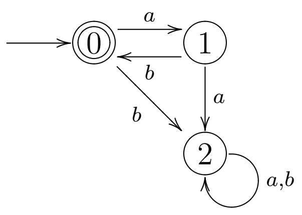
   The minimal automaton is therefore



   (iii)
   - $\varepsilon^{-1}L = L$.
   - $a^{-1}L = b(ab)^* = L_1$.
   - $b^{-1}L = \emptyset = L_2$.

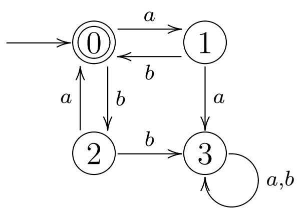
- $a^{-1}L_1 = \emptyset = L_2$.
- $b^{-1}L_1 = L$.
- $a^{-1}L_2 = L_2$.
- $b^{-1}L_2 = L_2$.

The minimal automaton is therefore



(iv)
- $\varepsilon^{-1}L = L$.
- $a^{-1}L = bL = L_1$.
- $b^{-1}L = aL = L_2$.
- $a^{-1}L_1 = \emptyset = L_3$.
- $b^{-1}L_1 = L$.
- $a^{-1}L_2 = L$.
- $b^{-1}L_2 = \emptyset = L_3$.
- $a^{-1}L_3 = b^{-1}L_3 = L_3$.

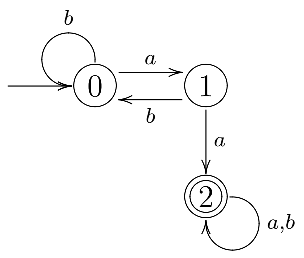
The minimal automaton is therefore



(v)
- $\varepsilon^{-1}L = L$.
- $a^{-1}L = (a+b)^*a^2(a+b)^* + a(a+b)^* = L_1$.
- $b^{-1}L = (a+b)^*a^2(a+b)^* = L$.
- $a^{-1}L_1 = (a+b)^* = L_2$.
- $b^{-1}L_1 = L$.
- $a^{-1}L_2 = L_2$.

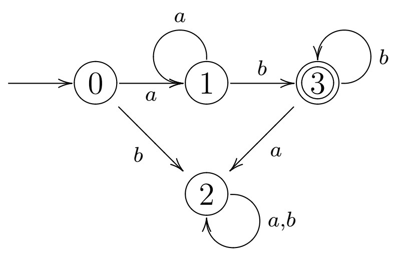
- $b^{-1}L_2 = L_2$.

The minimal automaton is therefore



(vi)
- $\varepsilon^{-1}L = L$.
- $a^{-1}L = a^*bb^* = L_1$.
- $b^{-1}L = \emptyset = L_2$.
- $a^{-1}L_1 = L_1$.
- $b^{-1}L_1 = b^* = L_3$.
- $a^{-1}L_2 = \emptyset = L_2$.
- $b^{-1}L_2 = \emptyset = L_2$.
- $a^{-1}L_3 = \emptyset = L_2$.
- $b^{-1}L_3 = b^* = L_3$.

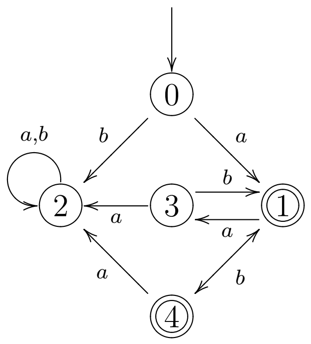
The minimal automaton is therefore



(vii)
- $\varepsilon^{-1}L = L$.
- $a^{-1}L = (b^2 + ab)^*b^* = L_1$.
- $b^{-1}L = \emptyset = L_2$.
- $a^{-1}L_1 = b(b^2 + ab)^*b^* = L_3$.
- $b^{-1}L_1 = b(b^2 + ab)^*b^* + b^* = L_4$.
- $a^{-1}L_2 = L_2 = b^{-1}L_2$.
- $a^{-1}L_3 = \emptyset = L_2$.

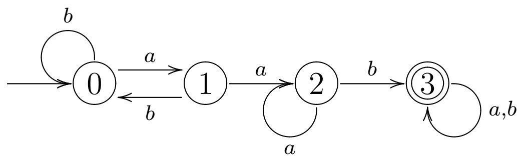
- $b^{-1}L_3 = (b^2 + ab)^*b^* = L_1$.
- $a^{-1}L_4 = \emptyset = L_2$.
- $b^{-1}L_4 = (b^2 + ab)^*b^* = L_1$.

The minimal automaton is therefore



(viii)
- $\varepsilon^{-1}L = L$.
- $a^{-1}L = L + ab(a+b)^* = L_1$.
- $b^{-1}L = L$.
- $a^{-1}L_1 = L_1 + b(a+b)^* = L_2$.
- $b^{-1}L_1 = L$.
- $a^{-1}L_2 = L_2$.
- $b^{-1}L_2 = (a+b)^* = L_3$.
- $a^{-1}L_3 = L_3 = b^{-1}L_3$.

The minimal automaton is therefore

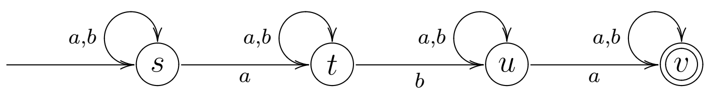


4. The quotients are: $\emptyset$, $L_r = \{a^{n-r}b^n \colon n \geq r\}$, and b^r for $r = 0, 1, 2, \ldots$.

2008 Exam paper

Attempt 3 questions in 2 hours

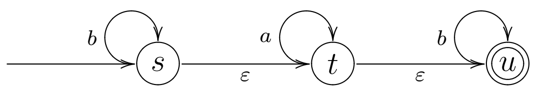
1. (i) Write down a regular expression describing the language L recognised by the non-deterministic automaton below.



   (ii) Apply the *accessible* subset construction to the machine in (i), and so find a deterministic automaton recognising the language L. To obtain full credit, you must show all steps in the algorithm.

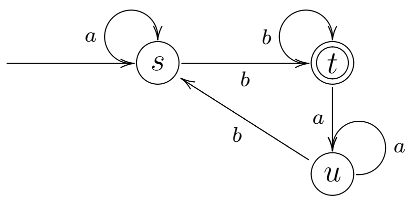
   (iii) Apply the minimisation algorithm to the automaton constructed in (ii), and so find the minimal automaton recognising the language L. To obtain full credit, you must show all steps in the algorithm.

2. (i) Write down a regular expression describing the language L accepted by the following ε-automaton.



   (ii) Apply the standard algorithm to the machine in (i) that converts it into a non-deterministic automaton without ε-transitions recognising the language L. To obtain full credit, you must show all steps in the algorithm.

   (iii) Apply the standard algorithm to the machine below to determine a regular expression for the language the machine recognises. To obtain full credit, you must show all steps in the algorithm.



3. (i) Define what is meant by a *regular expression* over the alphabet $A = \{a, b\}$, and a *regular language* over the alphabet A.

   (ii) State, without proof, Kleene's Theorem.

   (iii) Prove that the language $L = \{a^n b^n \colon n \geq 0\}$ is not recognisable.

   (iv) Prove that if L and M are both recognisable languages then $L \cap M$ is recognisable.

   (v) Prove that the language $M = \{x \in (a + b)^* \colon |x|_a = |x|_b\}$ is not recognisable.

4. (i) Let $\mathbf{A}$ be a complete deterministic automaton with input alphabet A. Define the *indistinguishability relation* $\simeq_{\mathbf{A}}$ on $\mathbf{A}$.

   (ii) Prove that $\simeq_{\mathbf{A}}$ is an equivalence relation on the set of states of $\mathbf{A}$, and that for each letter $a \in A$, we have that $s \simeq_{\mathbf{A}} t$ implies that $s \cdot a \simeq_{\mathbf{A}} t \cdot a$.

   (iii) Prove that if $s \simeq_{\mathbf{A}} t$ then s is terminal if and only if t is terminal.

   (iv) What does it mean to say that an automaton is *reduced*?

   (v) Prove that for each complete deterministic automaton $\mathbf{A}$, there is a reduced, complete deterministic automaton that recognises the same language as $\mathbf{A}$.

Solutions to 2008 exam

Below you will find outline solutions. Towards the end of the module, I will go through the solutions in more detail giving you an opportunity to try the questions first and to ask questions about them.

The rough format of the exam is two purely algroithmic questions and two theory questions.

However, it is important to remember that the next exam will not simply be the same questions with the numbers changed. The idea of giving you a sample exam paper with solutions is to show you the sort of things you might be asked: it is about getting a sense of the style of the questions.

1. (i) $(a+b)^*a(a+b)^*b(a+b)^*a(a+b)^*$. [2 marks]

   (ii) The first step is to construct the transition tree. Because this almost completely solves the problem you get [7 marks] for this part of the question.

$$\begin{tikzcd}
 & & s \arrow[dl, "a"'] \arrow[dr, "b"] & \\
 & s,t \arrow[dl, "a"'] \arrow[dr, "b"] & & s,\times \\
s,t,\times & & s,t,u \arrow[dl, "a"'] \arrow[dr, "b"] & \\
 & s,t,u,v \arrow[dl, "a"'] \arrow[dr, "b"] & & s,t,u,\times \\
s,t,u,v,\times & & s,t,u,v,\times &
\end{tikzcd}$$

You then have to carry out the glueing part of the procedure to obtain the required complete deterministic automaton. This gets you an additional [2 marks]. The resulting machine has the following form

$$\begin{tikzcd} {} \arrow[r] & 1 \arrow[r, "a"] \arrow[loop above, "b"] & 2 \arrow[r, "b"] \arrow[loop above, "a"] & 3 \arrow[r, "a"] \arrow[loop above, "b"] & 4 \arrow[loop above, "{a,b}"] \end{tikzcd}$$

(iii) The table that results in carrying out the minimisation algorithim is

	s_1	s_2	s_3	s_4
s_1	$\surd$	X	X	X
s_2	$*$	$\surd$	X	X
s_3	$*$	$*$	$\surd$	X
s_4	$*$	$*$	$*$	$\surd$

It follows that the associated equivalence classes are

$$\{s_1\}, \{s_2\}, \{s_3\}, \{s_4\}.$$

Thus the machine is already reduced. The marks are awarded as follows: [2 marks] for initialization, [2 marks] for the correct application of the algorithm and [5 marks] for the correct equivalence classes.

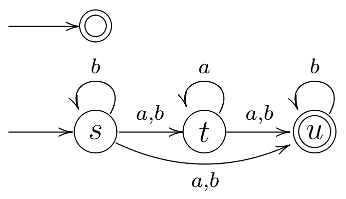
2. (i) $b^*a^*b^*$. [2 marks]

(ii) We first construct the table by applying the appropriate algorithm. This is the key to the whole procedure and for it you get [6 marks].

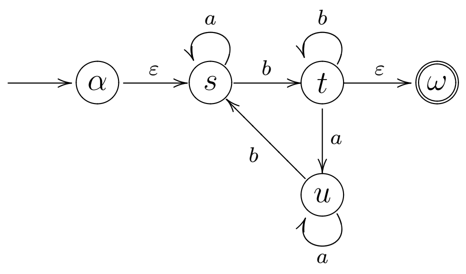
$*$	$E(*)$	$E(*) \cdot a$	$E(*) \cdot b$	$E(E(*) \cdot a)$	$E(E(*) \cdot b)$
s	$\{s, t, u\}$	$\{t\}$	$\{s, u\}$	$\{t, u\}$	$\{s, t, u\}$
t	$\{t, u\}$	$\{t\}$	$\{u\}$	$\{t, u\}$	$\{u\}$
u	$\{u\}$	$\emptyset$	$\{u\}$	$\emptyset$	$\{u\}$

The machine we want is constructed from the last two columns of this table, when there are 2 inputs. However, in this case there will be an extra state to recognise the empty string. There are

[3 marks] for the machine itself.

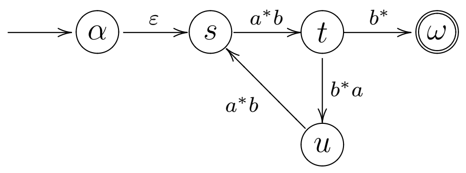


(iii) The first step is to normalise the machine. If you don't do this you will automatically get zero.



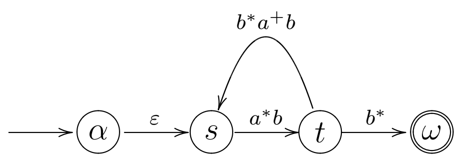
Each step of the algorithm should be shown making clear what you are doing at each stage.

First we eliminate loops.

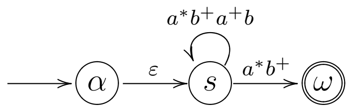


From this point on there are a number of different routes to a correct solution.

Eliminate state u.



Eliminate state t.



Eliminating loops and then state s we get

$$\begin{tikzcd} {} \arrow[r] & \alpha \arrow[r, "(a^*b^+a^+b)^*a^*b^+"] & \omega \end{tikzcd}$$

Thus

$$L(\mathbf{A}) = (a^*b^+a^+b)^*a^*b^+.$$

[9 marks]

3. (i) a and b are regular expressions, as are $\emptyset$ and ε; if r and s are regular expressions so too are $r + s$, rs and r^*; every regular expression is obtained in this way. [2 marks]

   A language is regular if it is described by means of a regular expression. [1 mark]

   (ii) A language is recognisable if and only if it is regular. [2 marks]

   (iii) This is a standard piece of bookwork. Suppose that L were recognised by the machine $\mathbf{A}$ with initial state i. Define $q_m = i \cdot a^m$. Suppose that $q_m = q_n$ for some $m \neq n$. Then $a^m b^n \in L(\mathbf{A})$ but $a^m b^n \notin L$. This is a contradiction and so L is not recognisable. [5 marks]

   (iv) Let $L = L(\mathbf{A})$ and $M = L(\mathbf{B})$ where $\mathbf{A} = (S, A, s_0, \gamma, F)$ and $\mathbf{B} = (Q, S, q_0, \delta, G)$. Put

$$\mathbf{A} \times \mathbf{B} = (S \times Q, A, (s_0, q_0), \gamma \times \delta, F \times G),$$

   a finite state automaton. [2 marks]

   We now check that $x \in L(\mathbf{A} \times \mathbf{B})$ iff $x \in L \cap M$. [4 marks] We have that $x \in L(\mathbf{A} \times \mathbf{B})$ iff $(s_0, q_0) \cdot x$ is terminal iff $(s_0 \cdot x, q_0 \cdot x)$ is terminal iff $s_0 \cdot x$ and $q_0 \cdot x$ are both terminal iff $x \in L$ and $x \in M$ iff $x \in L \cap M$, as required.

   (v) Suppose that M is recognisable. The language a^*b^* is recognisable by Kleene's Theorem. Thus by (iv), we have that $a^*b^* \cap M$ is recognisable. But $L = a^*b^* \cap M$, which is not recognisable by (iii) above. Contradiction, and so M is not recognisable. [4 marks]

4. (i) Let s and t be a pair of states. We say that $s \simeq_{\mathbf{A}} t$ if and only if for all $x \in A^*$ we have that $s \cdot x$ is terminal iff $t \cdot x$ is terminal. [1 mark]

(ii) We have to show that $\simeq_{\mathbf{A}}$ is reflexive, symmetric and transitive. [1 mark] will be awarded for each of these.

Suppose that $s \simeq_{\mathbf{A}} t$. Let $a \in A$. Suppose that $(s \cdot a) \cdot x$ is terminal. Then $s \cdot (ax)$ is terminal. Thus by assumption, $t \cdot (ax)$ is terminal and so $(t \cdot a) \cdot x$ is terminal. The converse is also true and so we have proved that $s \cdot a \simeq_{\mathbf{A}} t \cdot a$. [3 marks]

(iii) Suppose that s is terminal. Then $s \cdot \varepsilon$ is terminal, and so by assumption $t \cdot \varepsilon$ is terminal yielding t is terminal. The converse is proved similarly.

(iv) An automaton is reduced iff $\simeq_{\mathbf{A}}$ is the equality relation. [1 mark]

(v) Let $\mathbf{A} = (S, A, s_0, \delta, T)$ be an automaton. We define a machine $\mathbf{A}^r = \mathbf{A}/ \simeq_{\mathbf{A}}$ as follows. The set of states is the set of $\simeq_{\mathbf{A}}$-equivalence classes; I will denote the $\simeq_{\mathbf{A}}$-equivalence class containing the state s by $[s]$. The initial state is $[s_0]$. A state $[s]$ is terminal iff s is terminal. The transition function is given by $[s] \cdot a = [s \cdot a]$; this is well-defined by (ii). [2 marks]

We have to prove that $L(\mathbf{A}^r) = L(\mathbf{A})$. By induction, $[s] \cdot x = [s \cdot x]$. We have that $x \in L(\mathbf{A}^r)$ iff $[s_0] \cdot x$ is terminal iff $[s_0 \cdot x]$ is terminal iff by (iii) $s_0 \cdot x$ is terminal iff $x \in L(\mathbf{A})$, as required. [4 marks]

It remains to be proved that $\mathbf{A}^r$ is reduced. The states $[s]$ and $[t]$ are indistinguishable iff for all strings x we have that $[s] \cdot x$ is terminal iff $[t] \cdot x$ is terminal iff $[s \cdot x]$ is terminal iff $[t \cdot x]$ is terminal iff, by (iii), $s \cdot x$ is terminal iff $t \cdot x$ is terminal, which means precisely that s and t are indistinguishable in $\mathbf{A}$, and so $[s] = [t]$, as required. [4 marks]

Bibliography

[1] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: principles, techniques, and tools*, Addison-Wesley, 1986.

[2] M. A. Arbib, *Brains, machines and mathematics*, Springer-Verlag, 1987.

[3] M. P. Béal, D. Perrin, Symbolic dynamics and finite automata, in *Handbook of formal languages, Volume 2* (editors G. Rozenberg, A. Salomaa), Springer, 1997, 463–506.

[4] W. Brauer, *Automatentheorie*, B.G. Teubner, Stuttgart, 1984.

[5] J. Carroll, D. Long, *Theory of finite automata*, Prentice-Hall International, 1989.

[6] N. Chomsky, Three models for the description of languages, *IRE Transactions of Information Theory* **2** (1956), 113–124.

[7] P. S. Churchland, *Neurophilosophy*, The MIT Press, 1990.

[8] D. I. A. Cohen, *Introduction to computer theory*, Second Edition, John Wiley and Sons, 1997.

[9] M. Chrochemore, C. Hancart, Automata for matching patterns, in *Handbook of formal languages, Volume 2* (editors G. Rozenberg, A. Salomaa), Springer, 1997, 399–462.

[10] D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. F. Levy, M. S. Paterson, W. P. Thurston, *Word processing in groups*, Jones and Bartlett, 1992.

[11] J. E. F. Friedl, *Mastering regular expressions*, Second Edition, O'Reilly, 2002.

[12] F. Gécseg, I. Peák, *Algebraic theory of automata*, Akadémiai Kiadó, Budapest, 1972.

[13] V. M. Glushkov, The abstract theory of automata, *Russian Mathematical Surveys* **16** (1961), 1–53.

[14] R. I. Grogorchuk, V. V. Nekrashevich, V. I. Sushchanskii, Automata, dynamical systems, and groups, *Proceedings of the Steklov Institute of Mathematics* **231** (2000), 128–203.

[15] F. von Haeseler, *Automatic sequences*, Walter de Gruyter, 2003.

[16] A. Hodges, *Alan Turing: the enigma*, Vintage, 1992.

[17] J. E. Hopcroft, J. D. Ullman, *Introduction to automata theory, languages and computation*, Addison-Wesley, 1979.

[18] J. E. Hopcroft, R. Motwani, J. D. Ullman, *Introduction to automata theory, languages and computation*, Second Edition, Addison Wesley, 2001.

[19] D. A. Huffman, The synthesis of sequential switching circuits, *Journal of the Franklin Institute* **257** (1954), 161–190, 275–303.

[20] S. C. Kleene, Representation of events in nerve nets and finite automata, in *Automata studies* (editors C. E. Shannon, J. McCarthy), Princeton University Press, 1956, 3–42.

[21] D. C. Kozen, *Automata and computability*, Springer-Verlag, 1997.

[22] H. R. Lewis, C. H. Papadimitriou, *Elements of the theory of computation*, Second Edition, Addison Wesley Longman, 1998.

[23] D. Lind, B. Marcus, *Symbolic dynamics and coding*, Cambridge University Press, 1995.

[24] M. Lothaire, *Combinatorics on words*, Cambridge University Press, 1997.

[25] W. S. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, *Bulletin of Mathematical Biophysics* **5** (1943), 115–133.

[26] G. H. Mealy, A method for synthesizing sequential circuits, *Bell System Technical Journal* **34** (1955), 1045–1079.

[27] Yu. T. Medvedev, On the class of events representable in a finite automaton, 1956, in Russian, reprinted in English in [31].

[28] B. Mikolajczak (editor), *Algebraic and structural automata theory*, North-Holland, 1991.

[29] M. Minsky, *Computation: finite and infinite machines*, New York, Prentice-Hall, 1967.

[30] E. F. Moore, Gedanken-Experiments on sequential machines, in *Automata studies* (editors C. E. Shannon, J. McCarthy), Princeton University Press, 1956, 129–153.

[31] E. F. Moore (editor), *Sequential machines: selected papers*, Addison-Wesley, 1964.

[32] J. Myhill, Finite automata amd the representation of events, *Wright Air Development Command Technical Report* **57–624**, (1957), 112–137.

[33] A. Nerode, Linear automaton transformations, *Proceedings of the American Mathematical Society* **9** (1958), 541–544.

[34] D. Perrin, Finite automata, in *Handbook of theoretical computer science* (editor J. van Leeuwen), Elsevier Science Publishers B.V., 1990, 3–57.

[35] D. Perrin, Les débuts de la theorie des automates, *Technique et Science Informatique* **14** (1995), 409–433.

[36] C. Petzold, *Codes*, Microsoft Press, 1999.

[37] J.-E. Pin, *Varieties of formal languages*, North Oxford Academic, 1986.

[38] M. O. Rabin, D. Scott, Finite automata and their decision problems, *IBM Journal of Research and Development* **3** (1959), 114–125. Reprinted in *Sequential machines* (editor E. F. Moore), Addison-Wesley, Reading, Massachusetts, 1964, 63–91.

[39] E. Roche, Y. Schabes (editors), *Finite-state language processing*, The MIT Press, 1997.

[40] A. Salomaa, *Theory of automata*, Pergamon Press, 1969.

[41] M. P. Schützenberger, Une théorie algébrique du codage, in *Séminaire Dubreil-Pisot* (1955/56), exposé no. 15.

[42] M. P. Schützenberger, Une théorie algébrique du codage, *Comptes Rendus des Séances de l'Académie des Sciences Paris* **242** (1956), 862–864.

[43] C. E. Shannon, J. McCarthy (editors), *Automata studies*, Princeton University Press, Princeton, New Jersey, 1956.

[44] D. Shasha, C. Lazere, *Out of their minds*, Copernicus, 1995.

[45] C. C. Sims, *Computation with finitely presented groups*, Cambridge University Press, 1994.

[46] M. Sipser, *Introduction to the theory of computation*, PWS Publishing Company, 1997.

[47] M. Smith, *Station X*, Channel 4 Books, 2000.

[48] W. P. Thurston, Groups, tilings and finite state automata, *Summer 1989 AMS Colloquium Lectures*, National Science Foundation, University of Minnesota.

[49] A. M. Turing, On computable numbers with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* **2** (1936), 230–265. Erratum: Ibid **43** (1937), 544–546.

Index