

COMP20121  
The Implementation and Power of Computer Languages  
(The ‘Power’ Part of the Course)

Andrea Schalk  
A.Schalk@cs.man.ac.uk  
Department of Computer Science  
University of Manchester

September 26, 2006

**Text books.** Any text book will provide a much more detailed account of this topic. It will give many more examples and exercises, and will often go beyond what is taught in this course (and beyond what is examinable). If you are trying to learn the material by yourself you are well-advised to combine these notes with a text book, using the former to tell you which chapters you have to understand. When revising, reading a different account of the same material can help to fend off boredom, or just test your understanding. For suggestions see the list of references below. Any introductory text on automata theory will cover all that is required for Sections 1–2, although you may have to find another book for Sections 3–4—any account of computability will probably go beyond what we aim to do here.

## References

- [HMU01] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition edition, 2001. ISBN 0-201-44124-1.
- [Koz97] D. Kozen. *Automata and Computability*. Springer, 1997. ISBN 0-387-94907-0.
- [Sip97] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997. ISBN 0-534-94728-X.

**Acknowledgements.** My thanks to Peter Aczel, Don MacInnes, Matthew Collinson, Harold Simmons, Pete Jinks and Roy Schestowitz for suggesting improvements to these notes. Thanks also to Achim Jung for giving me access to his notes. Some of the presentation and some of the exercises come from there.

# Contents

<b>1</b>	<b>Regular Languages and Finite Automata</b>	<b>3</b>
1.1	Some preliminaries: Alphabets, words, languages . . . . .	3
1.2	Regular expressions and regular languages . . . . .	5
1.3	Finite automata . . . . .	7
1.4	Languages defined by DFAs . . . . .	11
1.5	Finite automata for regular expressions . . . . .	14
1.6	From NFAs to DFAs . . . . .	17
1.7	Properties of regular languages . . . . .	20
1.8	Limitations of regular languages . . . . .	21
1.9	Exercises for Section 1 . . . . .	23
<b>2</b>	<b>Context-free Grammars and Pushdown Automata</b>	<b>27</b>
2.1	Context-free grammars . . . . .	27
2.2	Pushdown automata . . . . .	30
2.3	From context-free grammars to PDAs . . . . .	36
2.4	Context-free grammars and PDAs . . . . .	38
2.5	Deterministic and non-deterministic PDAs . . . . .	39
2.6	Limitations of context-free languages . . . . .	39
2.7	Exercises for Section 2 . . . . .	42
<b>3</b>	<b>Turing Machines</b>	<b>44</b>
3.1	Automata with random-access memory—Turing machines . . . . .	44
3.2	Languages accepted by Turing machines . . . . .	48
3.3	Chomsky’s hierarchy of grammars . . . . .	51
3.4	Exercises for Section 3 . . . . .	54
<b>4</b>	<b>Computability</b>	<b>56</b>
4.1	Turing machine and computations . . . . .	56
4.2	Variants of Turing machines . . . . .	58
4.3	The power of Turing machines—the Church-Turing Thesis . . . . .	61
4.4	The Halting Problem . . . . .	62
4.5	Turing machines and computability . . . . .	64
4.6	Some decidable problems . . . . .	67
4.7	Some undecidable problems . . . . .	68
4.8	Applications in mathematics . . . . .	70
4.9	Exercises for Section 4 . . . . .	72

# 1 Regular Languages and Finite Automata

In this section we introduce some basic concepts, such as alphabet, word and language. We then introduce the notion of a *pattern*, which allows us to describe a number of words at the same time, namely those which *match the pattern*. Patterns are easy to handle for computers, but not so nice for humans, and we introduce certain *finite state automata* to help with that. We establish the connections between pattern and automata, and finally examine what kinds of languages we can describe using these.

## 1.1 Some preliminaries: Alphabets, words, languages

We begin by fixing notation and by defining a few simple concepts which are relevant for both parts of the course. This is a good time to point out that when we refer to the natural numbers,  $\mathbb{N}$ , we include 0.

**Definition 1** *An alphabet  $\Sigma$  is a finite set of characters (primitive indecomposable symbols) called the letters of  $\Sigma$ .*

We will normally use  $x$ ,  $y$  and  $z$  for unspecified elements of an alphabet. There are certain characters which we do not allow as letters because they have a meaning reserved for other purposes. The reserved characters we will use are listed in Table 1.

$\epsilon$	the empty word
$\epsilon$	the pattern for the empty word (note the bold face!)
$\emptyset$	the empty pattern (note the bold face!)
$\tau$	a move without action, a ‘silent’ move
$ , *$	pattern constructors
$(, )$	delineators
EOF	‘end of file’, sometimes needed to mark end of input
EOS	‘end of stack’, needed to mark that a stack is empty
$\sqcup$	a blank cell

Table 1: Reserved symbols and their meaning

Notice that in programming languages, strings are often considered to be primitives, or single letters, for example `if`, `while`, and the like.

**Definition 2** *A word over an alphabet  $\Sigma$  is a finite string*

$$x_1x_2 \cdots x_n$$

where the  $x_i$  are letters from  $\Sigma$ . This includes the empty string  $\epsilon$ . (It is made up of 0 letters, so  $n = 0$  for  $x_1 \cdots x_n$  in this case.)

We use  $\alpha$ ,  $\beta$ ,  $\gamma$  to refer to unspecified words over a given alphabet. We will later be forced to come up with a way of marking the end of a word, for which we will use the symbol EOF. We will therefore sometimes write a word as

$$x_1x_2 \cdots x_n \text{ EOF.}$$

However, this is not really required for the material presented in this section, and therefore we will not use this just yet. We can combine words by concatenating them. Note that concatenating the empty word with any other word gives that other word, that is we have that

$$\epsilon\alpha = \alpha = \alpha\epsilon.$$

We use  $\Sigma^*$  to refer to the set of all words over  $\Sigma$ . We find expressions more readable if we place  $*$  as a superscript when applied to sets, but as a postfix operator when applied to strings.

**Definition 3** *A language over an alphabet  $\Sigma$  is any subset of the set  $\Sigma^*$  of finite words over  $\Sigma$ .*

There is a largest language over  $\Sigma$ , namely all of  $\Sigma^*$ , as well as a smallest one: the empty language  $\emptyset$ . Note that while every word over an alphabet has finite length, there are *infinitely* many of those for any non-empty alphabet, and so languages can be infinite.

$\Sigma$	an alphabet
$x, y, z$	letters from the alphabet $\Sigma$
$\alpha, \beta, \gamma$	words over $\Sigma$
$L, L'$	languages over $\Sigma$
$p, p'$	patterns
$N$	non-terminal symbols (placeholders)

Table 2: Use of placeholders/variables

Note that we can perform the usual set operations with languages over the same alphabet. Let  $L$  and  $M$  be languages over the alphabet  $\Sigma$ . Then we can form

$$L \cup M, \quad L \cap M \quad \text{and} \quad \bar{L} = \Sigma^* \setminus L.$$

There are other things we can do by viewing such a language as giving us a new alphabet. Given a language  $L_1$  over the alphabet  $\Sigma_1$  and a language  $L_2$  over the alphabet  $\Sigma_2$  we can define the language  $L_1 \cdot L_2$  over  $\Sigma_1 \cup \Sigma_2$  by taking all words that are generated by taking a word from language  $L_1$  and appending a word from Language  $L_2$ . In other words

$$L_1 \cdot L_2 = \{\alpha_1\alpha_2 \mid \alpha_1 \in L_1, \alpha_2 \in L_2\}.$$

Similarly we can define  $L^n$  to be the set of all strings consisting of exactly  $n$  concatenated words from  $L$ . This includes  $L^0$ , the set of all words of length 0, which is nothing but  $\{\epsilon\}$ . If we define  $L^*$  to consist of all finite concatenations of words from  $L$ , so

$$L^* = \{\alpha_1 \cdots \alpha_n \mid n \in \mathbb{N}, \alpha_1, \dots, \alpha_n \in L\},$$

then  $L^* = \bigcup_{n \in \mathbb{N}} L^n$ . We can *reverse* all the words in  $L$  (read them from right to left) to obtain the language  $L^R$ , so

$$L^R = \{x_n x_{n-1} \cdots x_1 \mid x_1 x_2 \cdots x_n \in L\}.$$

## 1.2 Regular expressions and regular languages

Often when we search a file for something we do not look for just one specific word: For example, we might want to get all occurrences whether or not they are capitalized. If the word is a noun, we may want to search for it or its plural, and if it is a verb we may want to find all forms of it. *Regular expressions* supply a commonly used way of expressing such *search patterns*. You will see more applications of this idea in the other part of the course. Regular expressions are patterns built following certain rules.

**Definition 4** Let  $\Sigma$  be an alphabet. A **pattern** or **regular expression** over  $\Sigma$  is any word over

$$\Sigma^{\text{pat}} = \Sigma \cup \{\emptyset, \epsilon, |, *, (, )\}$$

generated by the following inductive definition.

**Empty pattern** The character  $\emptyset$  is a pattern;

**Empty word** the character  $\epsilon$  is a pattern;

**Base case** every letter from  $\Sigma$  is a pattern;

**Concatenation** if  $p_1$  and  $p_2$  are patterns then so is  $(p_1p_2)$ ;

**Alternative** if  $p_1$  and  $p_2$  are patterns then so is  $(p_1|p_2)$ ;

**Kleene star** if  $p$  is a pattern then so is  $(p^*)$ .

Most implementations of regular expressions (such as **grep** in Linux or Unix) allow more patterns than we have given here. For example, in **grep**, if  $p$  is a pattern then so is  $p+$  (see Exercise 2 (a) for the meaning of that pattern). However, we can express precisely the same languages with either form of pattern, that is, they have the same *power of expressivity*. As is standard when considering the theory of any structure, it makes sense to choose the *smallest* collection of patterns which does the job. Be warned that when I ask you to come up with regular expressions or patterns, (including in the exam!) I expect you to use this definition. I may deduct points if you use constructions other than the ones appearing in this definition.

Clearly not all strings over the alphabet  $\Sigma^{\text{pat}}$  are patterns. For example

$$)a|*(b$$

is not a pattern because it is not generated according to these rules. Because of the use of brackets, each pattern can be *uniquely parsed*, that is there is precisely one sequence of applying the above rules which leads to that pattern. That makes it fairly easy to decide whether or not a given string is indeed a pattern.

Note that in this definition there is no meaning given to the concatenation of two patterns, or the Kleene star, or the vertical bar  $|$ . This only comes with the following definition which tells us when a given word *matches* a given pattern. It should be pointed out that to make such patterns more readable, brackets are often omitted, and the convention is adopted that the Kleene star binds the strongest, followed by concatenation. This makes alternative the weakest one, and the appropriate bracketing for  $a*b|cda$  is therefore  $((a*b)|(cda))$ .<sup>1</sup>

<sup>1</sup>It turns out that there is no difference in meaning between  $((ab)c)$  and  $(a(bc))$ , nor between  $((a|b)|c)$  and  $(a|(b|c))$ , and we feel free to leave out brackets of this kind.

Each pattern  $p$  over  $\Sigma$  defines a language  $L(p)$  over  $\Sigma$ , namely the set of all the words which match the pattern  $p$ .

**Definition 5** Let  $p$  be a pattern over an alphabet  $\Sigma$  and let  $\alpha$  be a word over  $\Sigma$ . We say that  $\alpha$  **matches**  $p$  if one of the following cases holds:

**Empty word** The empty word  $\epsilon$  matches the pattern  $\epsilon$ ;

**Base case** the pattern  $p$  is a single character and  $\alpha$  is that character;

**Concatenation** the pattern  $p$  is a concatenation  $p = (p_1p_2)$  and there are words  $\alpha_1$  and  $\alpha_2$  such that  $\alpha_1$  matches  $p_1$ ,  $\alpha_2$  matches  $p_2$  and  $\alpha$  is the concatenation of  $\alpha_1$  and  $\alpha_2$ ;

**Alternative** the pattern  $p$  gives an alternative  $p = (p_1|p_2)$  and  $\alpha$  matches either  $p_1$  or  $p_2$  (it is allowed to match both);

**Kleene star** the pattern  $p$  is of the form  $p = (q^*)$  and  $\alpha$  can be written as a finite concatenation  $\alpha = \alpha_1\alpha_2\cdots\alpha_n$  such that  $\alpha_1, \alpha_2, \dots, \alpha_n$  all match  $q$ ; this includes the case where  $\alpha$  is empty (and thus an empty concatenation, with  $n = 0$ ).

Notice how the definition proceeds by recursion over the structure of the given pattern. Also note that there is no word which matches the empty pattern,  $\emptyset$ .

The pattern

$$a*b|cda$$

for example is matched by the words

$$b, ab, aab, aaab, cda.$$

It is *not* matched by the words

$$aabb, cd, abcda, abc.$$

You could try Exercises 1 and 2 now.

So for every pattern there are some words (possibly infinitely many) which match the pattern. All these words together define a language over the given alphabet  $\Sigma$  and we define

$$L(p) = \{\alpha \in \Sigma^* \mid \alpha \text{ matches } p\}.$$

**Definition 6** A language  $L$  is **regular** if it is the set of all words matching some regular expression, that is, if there is a pattern  $p$  such that  $L = L(p)$ .

Note that different patterns can define the same language, for example  $L(a^*) = L(\epsilon|aa^*)$ . Further note that for patterns  $p, p_1$  and  $p_2$  it is the case that

$$L(p_1|p_2) = L(p_1) \cup L(p_2) \quad \text{and} \quad L(p_1p_2) = L(p_1) \cdot L(p_2) \quad \text{and} \quad L(p^*) = L(p)^*.$$

The pattern  $\epsilon$  defines the language consisting of precisely the empty word, so  $L(\epsilon) = \{\epsilon\}$ , whereas the language recognized by the pattern  $\emptyset$  is the empty language,  $\emptyset$ . Further properties of regular languages are discussed in Section 1.7.

### 1.3 Finite automata

Computers are very good at manipulating symbols, deciding whether or not a word matches a pattern is the kind of task at which machines excel. Human beings, on the other hand, are somewhat different. Try writing down a moderately complicated pattern and a random word over the same alphabet, and then decide whether the word matches the pattern. You will find yourself trying to match up bits of the word with a part of the pattern, and notice that it is easy to lose track of where you were at a given moment. On the other hand, humans deal very well with pictorial representations of information, something computers have a hard time with (because it is difficult to describe a picture in the kind of language that computers understand—which all consist of long strings of symbols).

With a bit of experimenting, you might find yourself drawing pictures like that in Figure 1. Let us assume that we are considering words over the alphabet  $\Sigma = \{0, 1, 2, 3\}$  and that we are interested in the pattern  $((0^*)1)2|(0|(1^*))2$ . You might come up with the following, or something similar:

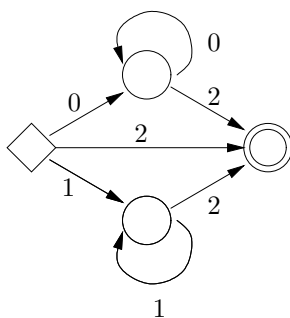


Figure 1: A state transition diagram

Now it seems much easier to decide whether or not the word 012 matches the pattern or not (it doesn't). The *meaning* of the picture seems intuitively clear. There's a starting position (the rhombus) and we read off the characters in the word one-by-one and match them against the characters labelling the arrows coming away from the 'current' position. We think of an arrow as a *transition* from one state to the next upon reading the appropriate input character. With the example, we have a first character 0 which takes us to the upper of the first two circles. There is no arrow labelled with 1 going from there, so we're stuck. On the other hand, if we consider 002 we can track a way from the starting position to the final position (the double circle), which means that this word is 'being accepted'.

Pictures like the above are called *state transition diagrams*. The underlying idea is quite simple. The rhombus and circles (including the double circle) stand for what are called 'states' - they help us to remember where we currently are. The arrows tell us how we can move from one state to the next by reading off the next symbol. In a picture we typically don't have to give the states any names, but if we want to talk about them it is much easier if we assume that they are named, that is, carry labels. Here is a formal definition of the kinds of diagrams we are interested in.

**Definition 7** Let  $\Sigma$  be a finite alphabet of symbols. A **(deterministic) finite automaton**, short **DFA**, over  $\Sigma$  consists of the following:

- A finite non-empty set  $Q$  of states;
- a particular element of  $Q$  called the start state (which we often denote with  $q_\bullet$ );
- a subset  $F$  of  $Q$  consisting of the accepting states;
- a transition function  $\delta$  which for every state  $q \in Q$  and every symbol  $x \in \Sigma$  returns the next state  $\delta(q, x) \in Q$ , so  $\delta$  is a function from  $Q \times \Sigma$  to  $Q$ . When  $\delta(q, x) = q'$  we often write

$$q \xrightarrow{x} q'.$$

We often put these four items together in a quadruple and speak of the DFA  $(Q, q_\bullet, F, \delta)$ .

The formal definition certainly looks somewhat more complicated than the almost self-explanatory picture in Figure 1. But that is just because we have a much better intuitive understanding of pictures than we do of formal descriptions.<sup>2</sup> There are a few differences between the formal definition and the picture in Figure 1 to which we will come in a moment. The description as ‘deterministic’ hints at the existence of ‘non-deterministic’ automata (see below). In a deterministic automaton, given a state  $q$  and a letter  $x$  from  $\Sigma$  there is precisely one arrow labelled with  $x$  from  $q$ . In other words, given  $x$  and  $q$  there is precisely one state where we can go from  $q$  by following the arrow labelled with  $x$ . This is the same as saying that  $\delta$  is a *function* (and not a relation).

**Definition 8** A **non-deterministic finite automaton**, short **NFA**, is given by a finite non-empty set  $Q$  of states, a start state in  $Q$  and a subset  $F$  of  $Q$  of accepting states as well as a transition relation  $\delta$  which relates a pair consisting of a state and a letter with a state. We often write

$$q \xrightarrow{x} q'$$

if  $(q, x)$  is  $\delta$ -related to  $q'$ .

Note that every deterministic automaton can be viewed as a non-deterministic automaton, but not the other way round. The relation between deterministic and non-deterministic finite automata is discussed in Section 1.5.

Now back to the example from Figure 1. When presenting DFAs pictorially, we can make use of different shapes of nodes to tell us whether they are ‘special’, that is whether they are the start state or an accepting state. In all these examples the start state will be given by a rhombus, and accepting states will be given by a double circle. In our example there is just one accepting state. However, the definition also says that *for all* states and letters, there should be an arrow labelled with that letter from that state. But our picture has no arrow labelled with 3 from the starting position (in fact, it has no arrows labelled with 3 at all). This is a convention often employed to make the picture easier to read. To adapt our picture to strictly satisfy the definition we would have to amend it to look like Figure 2.

We have added one ‘dump state’—we go there whenever we know that our word can’t possibly match the pattern, and once in that state, there is no escape—there is no way out of that state. Now for every state and every letter from  $\Sigma$  there is an arrow to some other state labelled with that letter. Where different letters lead to the same state we have used just one

---

<sup>2</sup>You might want to spend a few moments thinking about how you might implement a graph, or finite automaton. This kind of problem is at least briefly discussed in CS201.

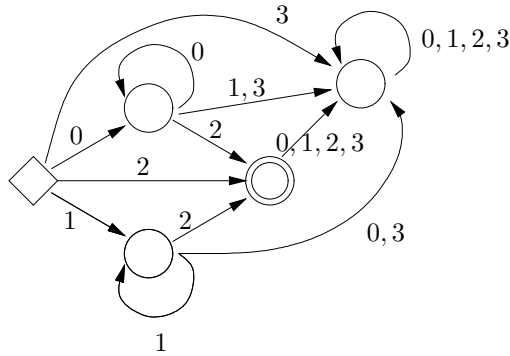


Figure 2: A picture describing a DFA

arrow and labelled it with all the appropriate letters to avoid cluttering up the picture even further.

Clearly there is no more information in Figure 2 than in Figure 1 and all we have done is to clutter up the picture, making it more difficult to read (for a person!). We will therefore adopt the convention that pictures such as that in Figure 1 are sufficient to describe the DFA given in Figure 2. To be more precise, from a picture like Figure 1 we obtain a DFA (satisfying the above definition) by adding one extra state and arrows to this state as follows: For every state, check whether there are any letters which do not have an arrow going away from that state. If this is the case add an arrow to the new state and label it with all the letters that are still missing. Finally, add an arrow from the new state to itself labelled with all the letters from the alphabet  $\Sigma$ . We refer to this new state as a ‘dump state’, because once we have reached it we cannot escape from it.

Finite automata are useful because just as we can say that a given word matches a given pattern, we can define when a finite automaton *accepts* a given word, which we sometimes call the ‘input string’ or the ‘input file’. Again we have to give a formal definition of what we mean by that, although intuitively this is quite clear.

**Definition 9** A word  $\alpha = x_1 \cdots x_n$  over  $\Sigma$  is accepted by the deterministic finite automaton  $(Q, q_\bullet, F, \delta)$  if  $\delta(q_\bullet, x_1) = q_1$ ,  $\delta(q_1, x_2) = q_2$ ,  $\dots$ ,  $\delta(q_{n-1}, x_n) = q_n$  and  $q_n \in F$ , that is,  $q_n$  is an accepting state. In particular, the empty word is accepted if and only if the start state is an accepting state.

A word  $\alpha = x_1 \cdots x_n$  over  $\Sigma$  is accepted by the non-deterministic finite automaton  $(Q, q_\bullet, \delta)$  if there are states  $q_0 = q_\bullet, q_1, \dots, q_n$  such that for all  $0 \leq i < n$ ,  $\delta$  relates  $(q_i, x_i)$  with  $q_{i+1}$  and  $q_n$  is an accepting state.

In either case we have a sequence of states  $q_\bullet, q_1, \dots, q_n$  such that

$$(q_\bullet = q_0) \xrightarrow{x_1} q_1 \xrightarrow{x_2} q_2 \longrightarrow \cdots \xrightarrow{x_n} q_n.$$

In particular, the empty word is accepted if and only if the start state is an accepting state.

**Definition 10** We say that a language is recognized by a finite automaton if it is the set of all words accepted by the automaton.

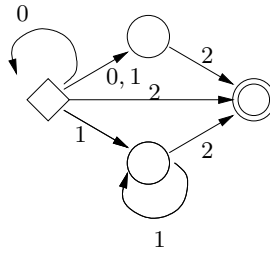


Figure 3: An NFA

If  $A$  is an automaton then we define  $L(A)$  to be the language it recognizes. To see whether a given word  $\alpha$  is accepted by the automaton  $A$  we begin at the initial state and trace out the unique path determined by  $\alpha$  ticking off each letter of  $\alpha$  as it is used. At each step we make the transition indicated by the current letter. When we have used up all of  $\alpha$  we have a final state  $q'$ . Then the word  $\alpha$  is accepted precisely when  $q'$  is an accepting state. This is much easier than seeing whether  $\alpha$  matches a given pattern.

We can use the same process with an automaton like that in Figure 1, but now we may reach a state with no exit before the whole word is used up. In that case the word is not accepted (even if that final state is an accepting state). This is because if we put in the ‘dump state’ introduced earlier then the next step takes us there, and we never escape. In this sense the two automata of Figure 1 and Figure 2 are equivalent. I will consider automata as in Figure 1 valid answers to exercises and exam questions. However, for some questions *you* may make your life easier by only considering automata that show all the states, as in Figure 2. Also note that sometimes we put a label on the states (the circles and the rhombus), and sometimes we don’t. Typically it is only worth doing if you need to refer to a state by its name, say in accompanying text.

The difference between a DFA and an NFA is the following. Compare the automaton in Figure 3 with that in Figure 1. There are two moves labelled with 0 from the start state, and two arrows labelled with 1. In order to decide whether the automaton in Figure 1 accepts the word 002 we just have to follow the string through the automaton, and the path we follow along the way is *unique*. But in order to decide whether the same string is accepted by the automaton in Figure 3, there are *several* paths through the automaton we have to consider. These are shown in Figure 4. To make it easier to follow where the automaton gets stuck, or ends its evaluation, the last state reached is filled in.

If *any* of these paths ends in an accepting state then the word is accepted by the automaton. You can think of the machine having to *guess* which transition it should follow in order to accept the given input. If all guesses fail then the string is not accepted.

So deterministic automata make it easier to decide whether a given word is accepted by such an automaton, but, as we shall see, it is often easier to construct a non-deterministic automaton satisfying some given criteria.

The automaton in Figure 5, for example, accepts the strings *abbbc* and *baaac*, but it does not accept *aabbc* or *abac*.

You now have everything you need to tackle Exercises 3–7 and Assessed Exercise 1.1.

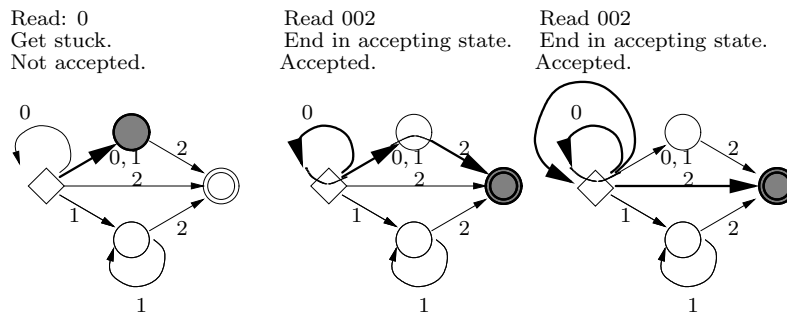


Figure 4: Word accepted?

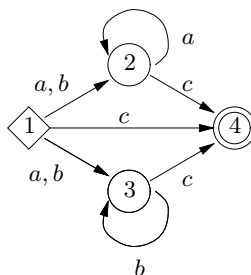


Figure 5: Another NFA

## 1.4 Languages defined by DFAs

To determine how useful finite automata are in dealing with regular expressions we have to consider the relation between the two concepts. We start by showing that if we are given a DFA then we can find a pattern such that the language recognized by the automaton is precisely the language of all words matching the pattern. In other words, DFAs are no more powerful than regular expressions. In the proof of the following proposition we actually describe an algorithm which, given a DFA, returns the corresponding pattern. Because this algorithm is general enough to apply to *all* DFAs, it looks fairly complicated.

**Proposition 1.1** *The language defined by a DFA is regular.*

**Proof sketch.** The proof of this is not quite straightforward and somewhat tedious. The algorithm is not one we would think of straight away because it has a little trick in it. We give a fairly detailed discussion of it below to illustrate how such a result can be proven rigorously. We do not treat all proofs at this level.

Let  $(Q, q_1, F, \delta)$  be a DFA (so the start state is  $q_1$ ) and let us assume that  $Q = \{q_1, \dots, q_n\}$ . The words accepted by this DFA are given by all the paths through the automaton which begin at the start state and end in an accepting state. So in order to find all the words accepted by the automaton we have to find all such paths. But because automata may contain loops, these paths can be arbitrarily long. What we need is a *systematic* way to describe *all* of them. Unfortunately, this leads to a fairly complicated notation. Let

$$P_{i \rightarrow j}^k$$

be the set of all paths from node  $q_i$  to node  $q_j$  such that all nodes ‘inside’ the path (that is all nodes with the exception of the end nodes) have an index less than or equal to  $k$ . For example  $q_4 \rightarrow q_2 \rightarrow q_1 \rightarrow q_5$  is an element of  $P_{4 \rightarrow 5}^2$  but not an element of  $P_{4 \rightarrow 5}^1$ . If we look at the words that label the paths in  $P_{i \rightarrow j}^k$  we can take this set of paths to define a language, which we will denote by  $L_{i \rightarrow j}^k$ .

If we look at a path that belongs to  $P_{i \rightarrow j}^k$  we notice that it may use  $q_k$  a few times (or not at all), and all other nodes occurring inside it will have indices strictly less than  $k$ . Hence we can either split up such a path into

- a path from  $q_i$  to  $q_k$  in  $P_{i \rightarrow k}^{k-1}$ ;
- several (or no) paths from  $q_k$  to  $q_k$  in  $P_{k \rightarrow k}^{k-1}$ ;
- a path from  $q_k$  to  $q_j$  in  $P_{k \rightarrow j}^{k-1}$ ;

or else if  $q_k$  does not occur at all in a given path then that path is (already) an element of  $P_{i \rightarrow j}^{k-1}$ . In other words we have that

$$L_{i \rightarrow j}^k = L_{i \rightarrow j}^{k-1} \cup (L_{i \rightarrow k}^{k-1} \cdot L_{k \rightarrow k}^{k-1*} \cdot L_{k \rightarrow j}^{k-1})$$

where we use the operations on languages introduced in Section 1.2. We thus get a set of recursion formulae. In the case where  $k = i$  or  $k = j$  we can simplify the formula to give

$$\begin{aligned} L_{i \rightarrow j}^i &= L_{i \rightarrow i}^{i-1*} \cdot L_{i \rightarrow j}^{i-1} && \text{and} \\ L_{i \rightarrow j}^j &= L_{i \rightarrow j}^{j-1} \cdot L_{j \rightarrow j}^{j-1*}. \end{aligned}$$

The language accepted by the automaton is the set of the words given by the paths starting in  $q_1$  and ending in a state that belongs to  $F$ , in other words it is

$$\bigcup_{q_l \in F} L_{1 \rightarrow l}^n.$$

We can use the recursive formulae to replace the  $L_{i \rightarrow j}^n$  in that formula, thus reducing the indices in the superscript until they are all equal to 0. The languages with superscript 0 are all fairly simple: If  $i \neq j$  then

- either  $L_{i \rightarrow j}^0$  is empty (if there is no arrow from  $q_i$  to  $q_j$ )
- or it consists of the letters labelling the unique arrow from  $q_i$  to  $q_j$ .

This is because such paths cannot have any inner nodes at all and therefore must have length one, so every word in some  $L_{i \rightarrow j}^0$  consists of at most one letter. Any  $L_{i \rightarrow i}^0$  will contain at least the empty word  $\epsilon$ , and if there is an arrow from  $q_i$  to itself, then it will contain all characters that label that arrow.

We will then end up with a fairly complicated expression of these  $L_{i \rightarrow j}^0$  combined with union ( $\cup$ ), concatenation ( $\cdot$ ), and the Kleene star ( $*$ )—applied to languages. Fortunately such expressions can be simplified according to the following rules:

- $\emptyset \cup L = L = L \cup \emptyset$ ;
- $\emptyset^* = \{\epsilon\}$ ;

- $\emptyset \cdot L = \emptyset = L \cdot \emptyset$ ;
- $(\{\epsilon\} \cup L)^* = L^* = (L \cup \{\epsilon\})^*$ ;
- $\{\epsilon\}^* = \{\epsilon\}$ ;
- $\{\epsilon\} \cdot L = L = L \cdot \{\epsilon\}$ .

Ultimately we can take the simplified expression consisting of some  $L^0$  and turn it into a regular expression by replacing the empty set with the empty pattern  $\emptyset$ , a set  $\{x_1, \dots, x_n\}$  with  $(x_1 | \dots | x_n)$ ,  $\cup$  with  $|$ , leaving out  $\cdot$  (since concatenation of patterns requires no symbol) and lowering  $*$  from a superscript to a postfix operator.  $\square$

To give an example we apply the algorithm to the automaton in Figure 6.

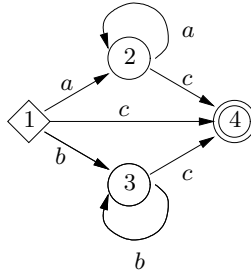


Figure 6: Find the language given by this automaton!

The following table gives the various  $L_{i \rightarrow j}^0$ .

$i \setminus j$	1	2	3	4
1	$\{\epsilon\}$	$\{a\}$	$\{b\}$	$\{c\}$
2	$\emptyset$	$\{\epsilon, a\}$	$\emptyset$	$\{c\}$
3	$\emptyset$	$\emptyset$	$\{\epsilon, b\}$	$\{c\}$
4	$\emptyset$	$\emptyset$	$\emptyset$	$\{\epsilon\}$

We calculate the language recognized by this automaton.

$$\begin{aligned}
L_{1 \rightarrow 4}^4 &= L_{1 \rightarrow 4}^3 \cdot L_{4 \rightarrow 4}^3^* \\
&= (L_{1 \rightarrow 4}^2 \cup (L_{1 \rightarrow 3}^2 \cdot L_{3 \rightarrow 3}^2 \cdot L_{3 \rightarrow 4}^2)) \cdot \{\epsilon\}^* \\
&= L_{1 \rightarrow 4}^1 \cup (L_{1 \rightarrow 2}^1 \cdot L_{2 \rightarrow 2}^1 \cdot L_{2 \rightarrow 4}^1) \cup ((L_{1 \rightarrow 3}^1 \cup (L_{1 \rightarrow 2}^1 \cdot L_{2 \rightarrow 2}^1 \cdot L_{2 \rightarrow 3}^1)) \cdot \{\epsilon, b\}^* \cdot \{c\}) \\
&= \{c\} \cup (\{a\} \cdot \{\epsilon, a\}^* \cdot \{c\}) \cup ((\{b\} \cup (\{a\} \cdot \{\epsilon, a\}^* \cdot \emptyset)) \cdot \{\epsilon, b\}^* \cdot \{c\}) \\
&= \{c\} \cup (\{a\} \cdot \{a\}^* \cdot \{c\}) \cup ((\{b\} \cup \emptyset) \cdot \{b\}^* \cdot \{c\}) \\
&= \{c\} \cup (\{a\} \cdot \{a\}^* \cdot \{c\}) \cup (\{b\} \cdot \{b\}^* \cdot \{c\})
\end{aligned}$$

Hence the desired pattern is

$$c|(aa^*c)|(bb^*c).$$

Now compare Figure 6 with Figure 1, and you will note that we were describing the ‘same’ automaton over the alphabet  $\{a, b, c\}$  rather than  $\{0, 1, 2\}$ . We originally started with pattern  $((0^*|1)2|(0|(1^*))2)$ , which is different from the one we have found now. Clearly there are rules for manipulating patterns without changing the accepted language. Maybe you can come up with some of those?

Beware that this very general algorithm typically gives large expressions for even moderately sized automata. It therefore pays at every step to simplify as much as possible! In particular it is often not necessary to go all the way down to superscript 0, since it may be possible to collect all paths from one node to another at an earlier stage, as in the above example.

If the automaton is small enough then, of course, one can just read off the corresponding pattern, and I will consider that a valid solution to exercises and exam questions (unless this algorithm was specifically asked for, as in Exercise 8). However, if you were to program this algorithm, or if the automaton you want to treat is moderately complex, there’s no way past this.

Try Exercise 8 and Assessed Exercise 1.2 now.

## 1.5 Finite automata for regular expressions

It remains to consider the converse of Proposition 1.1: Given a pattern, can we find a DFA which accepts precisely the language defined by the pattern?

**Proposition 1.2** *For every regular expression over some alphabet  $\Sigma$  there is a DFA that accepts precisely those words which match the regular expression. In other words: For every regular language there is a DFA which accepts precisely the words of that language.*

At a glance this looks as if it should be easy to prove. Since regular expressions are built inductively it should be possible to show that one can build a corresponding finite automaton by matching these inductive steps.

**Base case.** And the start seems quite encouraging: Clearly there is an automaton accepting precisely the empty word, and one accepting no word at all, and for every word consisting of precisely one character there is an automaton accepting precisely that word, see Figures 7 and 8.<sup>3</sup>

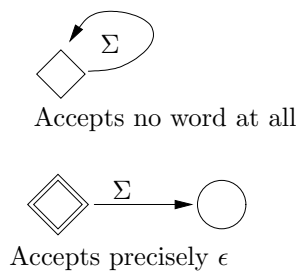


Figure 7: The empty word/pattern

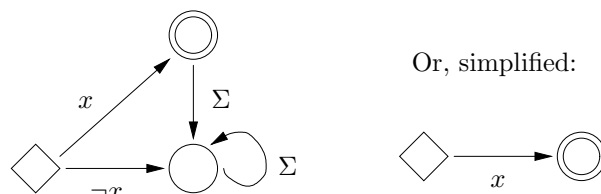


Figure 8: Accepts precisely the word  $x$

<sup>3</sup>We are introducing another convention here: an arrow labelled  $\neg x$  means one labelled with every symbol from the underlying alphabet *other than*  $x$ .

This takes care of the base case. The next operation is concatenation, and that's where it gets tricky. For example if we have a finite automaton  $A_1$  recognizing expressions matching some pattern  $p_1$ , and another finite automaton  $A_2$  recognizing expressions matching some pattern  $p_2$ , then one should be able to just 'put the second automaton after the first', that is, use every accepting state of  $A_1$  as the start state of a copy of the second automaton like this as in Figure 9.

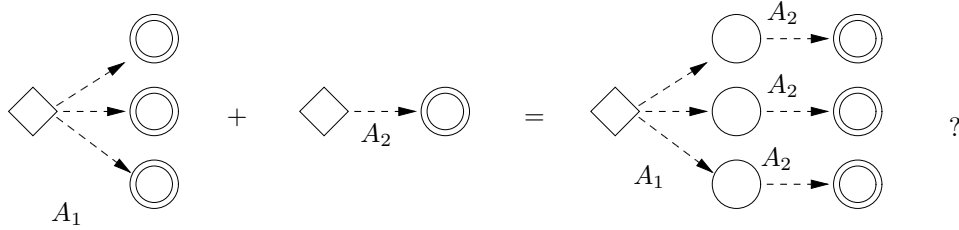


Figure 9: The composite of two DFAs?

The reason this doesn't work is that there maybe arrows back to the start state in both,  $A_1$  and  $A_2$ , which taken together might give a path from some state in  $A_2$  to the start state of  $A_1$ , and this does not fit with the way concatenation works. Take for example the pattern  $a*b*$  over the alphabet  $\{a, \dots, z\}$ . Both,  $a^*$  and  $b^*$  have very simple DFAs recognizing the pattern, but if we put them together in the proposed way then the resulting automaton will accept  $abab$ , which does not match the concatenated pattern  $a*b^*$ . (Note that this automaton is non-deterministic if the alphabet contains letters other than  $a$  and  $b$ !) This is demonstrated in Figure 10.

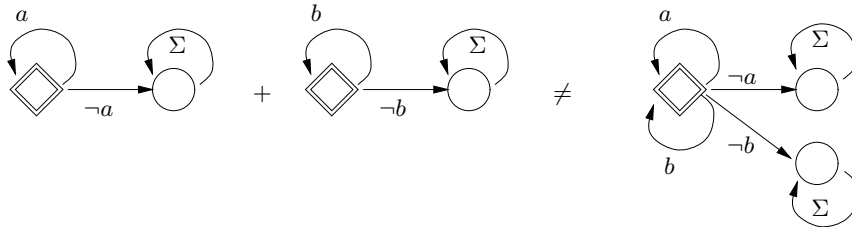


Figure 10: This composite is not what it should be

This means that we have to be somewhat more resourceful to prove this. The easiest way is to introduce an extra transition *which does not require any input symbol*. This allows us to keep the two automata we wish to combine 'at a distance' so that they can't interfere with each other. We do this formally by adding a new character to the alphabet  $\Sigma$  which we refer to as  $\tau$ .<sup>4</sup> We say that we have an **automaton with  $\tau$ -moves**, or **with  $\tau$ -transitions**. In order to check whether a word is accepted by such an automaton, all you have to know is that you may perform a  $\tau$ -move whenever you like; it is *not* matched to an input symbol.

**Concatenation.** If  $A_1$  is an automaton for pattern  $p_1$  and  $A_2$  is an automaton for pattern  $p_2$  then the following is an automaton for  $p_1p_2$ .

<sup>4</sup>This only works because in Table 1, we said that  $\tau$  is not allowed to occur in the alphabet  $\Sigma$ , just as  $\Sigma$  is not allowed to contain  $|$  or  $*$ .

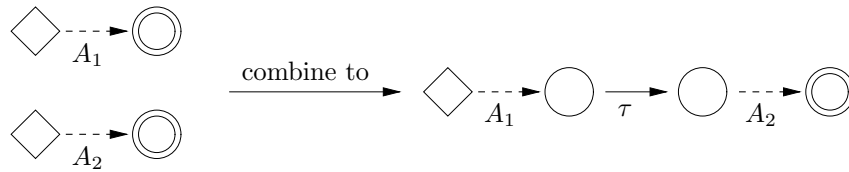


Figure 11: Concatenation

Note that the picture only shows one accepting state for  $A_1$ —to get this right, *for every accepting state* of  $A_1$ , we have to add a  $\tau$ -transition to the start state of  $A_2$ .

**Alternative.** The idea of  $\tau$ -transitions makes it quite easy to deal with this. If  $A_1$  is an automaton for pattern  $p_1$  and  $A_2$  is an automaton for pattern  $p_2$  then the following is an automaton for  $p_1|p_2$ .

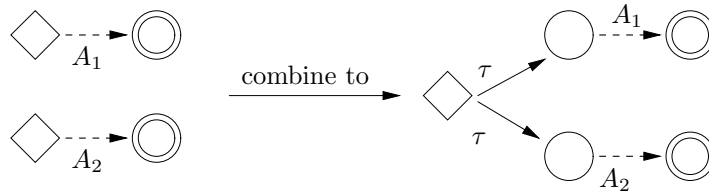


Figure 12: Alternative

In other words, we add a new start state which has  $\tau$ -transitions to the (now ‘old’) start states of  $A_1$  and  $A_2$ .

However, we notice that this is not a *deterministic* automaton any longer! We have two arrows leaving the start state which are both marked with the same symbol, namely  $\tau$ . We continue by trying to allow non-deterministic automata with  $\tau$ -moves for now and worry about the consequences later.

**Kleene star.** Finally, if  $A$  is an automaton for pattern  $p$  the the following is an automaton for pattern  $p^*$ .

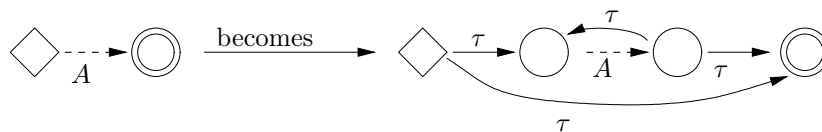


Figure 13: Kleene star

In words, we add a new start state which has a  $\tau$ -transition to the ‘old’ start state. Again, the picture shows only one accepting state. What we have to do is *for every ‘old’ accepting state* put in a  $\tau$ -transition to the ‘old’ start state. For every ‘old’ accepting state we then put in a new accepting state which is reached from that state by a  $\tau$ -move. We then make that ‘old’ accepting state non-accepting. Finally we add a  $\tau$ -transition from the new start state to every of the now existing accepting states, all of which are new.

So rather than proving Proposition 1.2, we have argued that the following holds.<sup>5</sup>

**Proposition 1.3** *For every regular expression over some alphabet  $\Sigma$  there is an NFA with  $\tau$ -transitions that accepts precisely those words which match the regular expression.*

**Proof sketch.** Convince yourself that this is true. We haven't given a formal proof, but if you understand the argument, you should be able to see that it can be turned into one. Exercise 9 provides an example.  $\square$

For an example, let us assume we wish to construct an automaton for the regular expression  $a^*b^*$ . This is carried out in Figure 14. Note how it avoids the problems encountered in Figure 10.

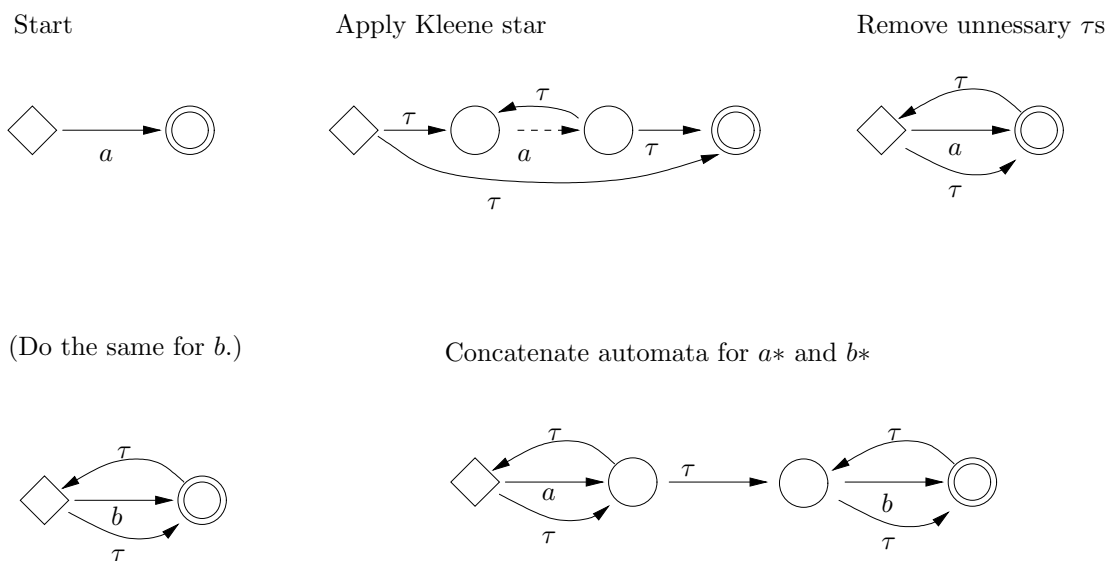


Figure 14: An automaton for  $a^*b^*$

Exercise 9 invites you to carry out the construction just introduced.

## 1.6 From NFAs to DFAs

But are NFAs really more powerful than DFAs? In other words, do we gain extra expressivity by allowing NFAs rather than DFAs to correspond to a given pattern? The answer is no, and we will prove this by showing the following.

**Proposition 1.4** *For every NFA with  $\tau$ -moves there exists a DFA that accepts precisely the same words.*

**$\tau$ -moves.** We show first of all that we can do without the  $\tau$ -transitions we introduced above. Let  $(Q, q_\bullet, F, \delta)$  be an NFA with  $\tau$ -transitions. We will define another automaton  $(Q, q_\bullet, F', \delta')$

<sup>5</sup>This is not a formal proof. The reason for this is not that we have been using pictures—rigorous proofs can be achieved with those, but that the pictures need some further explanation to make them unambiguous. When trying this out on an example it should be clear enough what is meant.

which does without the  $\tau$ -moves. This will be almost the same as the automaton we start with, having the same set of states and the same start state. The new transition relation  $\delta'$  and the new set of accepting states  $F'$  are obtained as follows.

- To obtain  $\delta'$ , take all transitions

$$q \xrightarrow{x} q'$$

for which there is a series of transitions from  $q$  to  $q'$  consisting of an arbitrary number of  $\tau$ -transitions (this may be 0  $\tau$ -transitions) followed by one last transition (to  $q'$ ) labelled with  $x$ , that is

$$q \xrightarrow{\tau} \dots \xrightarrow{\tau} q'' \xrightarrow{x} q'.$$

- To  $F'$ , add all states from which a state in  $F$  can be reached by by an arbitrary number of  $\tau$ -transitions. This describes  $F'$ .

We can ‘clean up’ the resulting automaton by removing states which are not reachable from the start state (since they cannot contribute to the set of accepted words in any way). Here is an example.

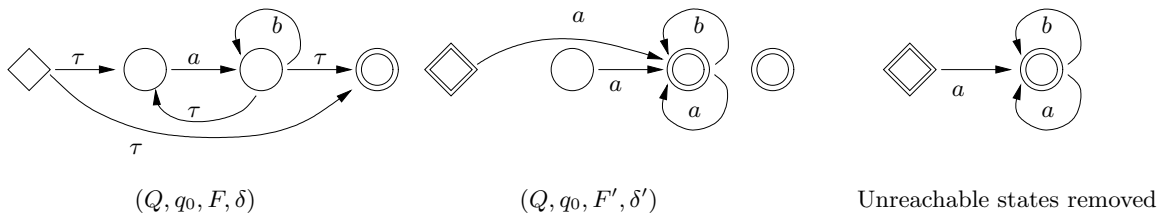


Figure 15: Removing  $\tau$ -transitions

Clearly even if the automaton we start with is deterministic these steps can easily lead to a non-deterministic one. This procedure works because it is the case that a word  $x_1 \cdots x_n$  is accepted by the new automaton if and only if there is a word  $\tau \cdots \tau x_1 \tau \cdots \tau x_2 \tau \cdots \tau x_n \tau \cdots \tau$  which is accepted by the original automaton. Exercise 10 invites you to try this algorithm.

**Non-determinacy.** Next we want to turn a non-deterministic automaton into a deterministic one. Let  $(Q, q_\bullet, F, \delta)$  be an NFA (without  $\tau$ -transitions). We want to construct a DFA which accepts precisely the same words. We do this by increasing the number of states considerably: The new set of states is  $\mathcal{P}(Q)$ , the powerset of the set of states of the original automaton. (Note that if  $Q$  is finite and non-empty then so is  $\mathcal{P}(Q)$ .) The start state is the singleton set  $\{q_\bullet\}$ . A set is an accepting state of the new automaton if it contains at least one state from  $F$ . It remains to describe the transition function. We do this by setting

$$\delta'(S, x) = \{q' \in Q \mid \exists q \in S. \text{ in } \delta, q \xrightarrow{x} q'\}.$$

In other words from a state  $S$  (which itself is a set of states from the old automaton) there is a transition labelled  $x$  to the set  $S'$  of all states which can be reached *from some* state in  $S$  by a transition labelled  $x$ . To obtain  $S'$ , go through all the states  $q$  in  $S$  and collect all the states  $q'$  for which there is a transition  $q \xrightarrow{x} q'$  in the old automaton, giving

$$S' := \{q' \in Q \mid \exists q \in S. \text{ in } \delta, q \xrightarrow{x} q'\}.$$

Again we can simplify the result by removing all states which cannot be reached from the start state. Note that  $\emptyset$  will always be a non-accepting state from which there is no escape, so it becomes a dump state and we can leave it out. An example is given in Figure 16.

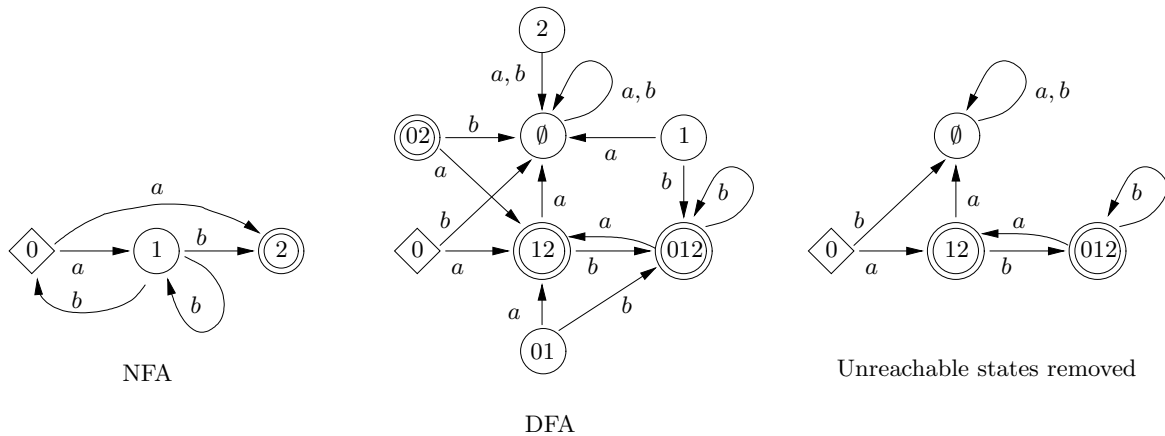


Figure 16: From non-determinism to determinism

This demonstrates that drawing the full automaton, with all states, goes to some unnecessary lengths: Surely it would be more sensible to generate only the reachable states in the first place? And, indeed, this can be done and is demonstrated in Figure 17. The trick is to begin at the start state  $\{q_\bullet\}$  (in Figure 17, that is  $\{0\}$ ) and then look at all the states that can be reached from there in one step. For this you need to consider every letter  $x$  of the alphabet  $\Sigma$  and see where the arrow labelled  $x$  ends up. There are arrows labelled  $a$  from 0 to states 1 and 2, and therefore for the new automaton From the start state  $\{0\}$  there is an arrow labelled  $a$  to the state  $\{1, 2\}$ . Since there is no arrow from the state 0 labelled  $b$ , the arrow labelled  $b$  from the state  $\{0\}$  ends in the empty set, but we argued above that we don't have to draw that. From the new state  $\{1, 2\}$  we have to consider transitions labelled with  $a$  from the states 1 and 2, but there aren't any, so there's nothing to draw. From states 1 and 2 there are arrows labelled  $b$  to all available states, so there is a transition labelled  $b$  to a new state,  $\{0, 1, 2\}$ . As it turns out, all transitions from that new state lead back to states which already exist. The process ends then because we have accounted for all transitions between the generated states. This is a technique known as breadth-first search. The resulting automaton is certainly much less complicated than the one we generated first!

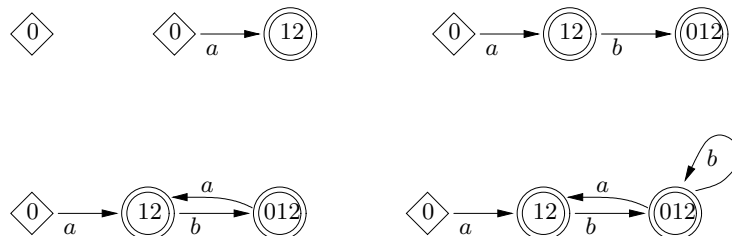


Figure 17: Generate only reachable states

It remains to show that the new deterministic automaton accepts precisely the same words as the non-deterministic one we started with. If a word is accepted by the original automaton then there exists a sequence of transitions from the start state labelled according to the word which ends in an accepting state. Clearly that means there exists a sequence of transitions from the new start state to a state which contains an accepting state of the original automaton and therefore is an accepting state of the new automaton. Hence all words accepted by the original automaton are accepted by the new automaton. On the other hand let us assume that a word is accepted by the new automaton. But then there must be a sequence of these transitions in the original automaton.

This (finally!) concludes the proof sketch of Proposition 1.2. The DFA constructed in this way will typically be fairly big—if the NFA has four states then the corresponding DFA might have as many as  $2^4$  states, unless only the reachable states are generated in the first place.

**Theorem 1.5** *A language is regular if and only if it is recognized by a deterministic finite automaton, and if and only if it is recognized by a non-deterministic automaton.*

In other words, there is a very strong relationship—(almost) as strong as it could be. It means that regular expressions are precisely as powerful in describing language as DFAs are.

You are now ready to have a go at Exercises 11 and 12 as well as Assessed Exercise 1.3. For some hard work, see Exercises 13.

## 1.7 Properties of regular languages

We note the following properties of regular languages.

**Proposition 1.6** (i) *Every finite language is regular.*

(ii) *If  $L$  is a regular language then so are  $L^n$  (for all  $n \in \mathbb{N}$ ),  $L^*$ ,  $\bar{L}$  and  $L^R$ .*

(iii) *If  $L$  and  $L'$  are regular languages then so are  $L \cup L'$ ,  $L \cap L'$  and  $L \cdot L'$ .*

**Proof.** (i) A finite language can be described using the alternative between all its words as the pattern, that is if  $L = \{\alpha_1, \dots, \alpha_n\}$  then a pattern describing  $L$  is given by  $\alpha_1 | \dots | \alpha_n$ . If  $L$  should be empty then it is regular by definition.

(ii) If  $p$  is a pattern for  $L$  then  $p \cdot \dots \cdot p$  with  $n$  copies of  $p$  is a pattern for  $L^n$ , and  $p^*$  is a pattern for  $L^*$ . In Assessed Exercise 1.1(b) you will show that if  $A$  is an automaton recognizing language  $L$  then we can give an automaton recognizing  $\bar{L}$ . By Theorem 1.5 that means that  $\bar{L}$  must be regular if  $L$  is. The last case is an exercise.

(iii) Assume that  $p$  is a pattern for  $L$  and that  $p'$  is a pattern for  $L'$ . Then  $p|p'$  is a pattern for  $L \cup L'$ , and  $pp'$  is a pattern for  $L \cdot L'$ . The remaining case is an exercise.  $\square$

Exercise 14 asks you to fill in the gaps in this proof.

## 1.8 Limitations of regular languages

So far regular languages were good enough for all that we tried. However, they are fairly limited. For example, there is no pattern (or automaton) which recognizes all the words over the alphabet  $\{0, 1\}$  which have as many 0s as 1s. Worse, the fairly restricted sublanguage  $\{0^n 1^n \mid n \in \mathbb{N}\}$  (where  $x^n$  stands for a word consisting of  $n$  times the character  $x$ ) is not regular.<sup>6</sup> The reason for this is that finite automata cannot count up to *arbitrarily large* numbers—the automaton would have to have a way of remembering how many 0s it has seen so far to know how many 1s must follow for a word to be accepted. The following Lemma states these limitations more generally. It applies to infinite languages. Looking at the way patterns are formed it becomes clear that there is only one way of generating infinitely many words which match a given pattern, and that is the use of the Kleene star. Hence we can generate words in that language by repeating the part that matches the pattern to which the Kleene star  $*$  applies. But if we only know the language, but not the pattern which describes it, how can we formulate this idea? This is done in the following lemma.

**Lemma 1.7 (The Pumping Lemma)** *Let  $L$  be an infinite regular language. Then there exists a natural number  $n > 0$  such that every word  $\alpha$  of  $L$  consisting of at least  $n$  characters contains a ‘pumping section’ in the following sense. The word  $\alpha$  can be split up into three parts, say  $\lambda$ ,  $\mu$ , and  $\nu$  such that*

- $\alpha = \lambda\mu\nu$ ;
- the ‘pumping section’  $\mu$  is not the empty word;
- the length of  $\lambda\mu$  is at most  $n$

and so that all words of the form

$$\lambda\mu^k\nu,$$

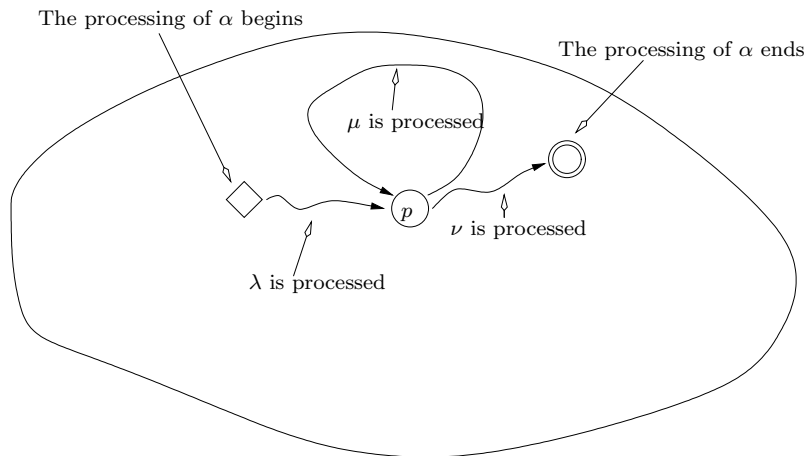
for  $k \in \mathbb{N}$ , belong to  $L$ .

**Proof sketch.** Let  $(Q, q_\bullet, F, \delta)$  be the finite automaton for  $L$ . We define  $n$  to be the number of states in  $Q$ . When processing a stretch  $\alpha$  of  $k$  letters the automaton will go through  $k + 1$  states (these letters define a path of length  $k$  which has  $k + 1$  nodes) not all of which have to be different. If  $\alpha$  has length greater than or equal to  $n$  then when processing the first  $n$  letters there is at least one state which is visited twice along this path. Call this state  $p$ . The situation is pictured in Figure 1.8. As indicated in Figure 1.8, we let  $\lambda$  be the string that is processed until  $p$  is reached for the first time. The string that is being processed from there until  $p$  is reached for the second time we choose as  $\mu$ . What remains of word  $\alpha$  we name  $\nu$ . The loop from  $p$  to  $p$ , processing  $\mu$ , clearly can be left out or repeated an arbitrary number of times without changing the final accepting state.  $\square$

Clearly there is no natural number  $n$  such that a word of shape  $0^i 1^i$  fits into this: Let us assume we had such an  $n$ . Consider the term  $0^n 1^n$ . Assume we have found strings  $\lambda$ ,  $\mu$  and  $\nu$  satisfying the conditions from the Pumping Lemma. Since  $\lambda\mu$  has length at most  $n$ ,  $\mu$

---

<sup>6</sup>Which means, of course, that the language  $\{a^n b^n \mid n \in \mathbb{N}\}$  is not regular either. It should be clear that changing the symbols of the alphabet does not make any real difference!



consists entirely of 0s, and at least one of those. That is, the situation is

$$\underbrace{00 \dots 00}_{\lambda} \underbrace{\dots 00}_{\mu} \underbrace{1 \dots 1}_n.$$

Hence  $\lambda\nu = \lambda\mu^0\nu$  looks like this

$$\underbrace{00 \dots 0}_{\lambda} \underbrace{1 \dots 1}_n$$

and therefore consists of more 1s than 0 and therefore is not in our language. Therefore such an  $n$  cannot exist, the language does not satisfy the Pumping Lemma, and thus cannot be regular. Informally, the reason for this is that no DFA (or NFA) can remember how many 0s it has seen to make sure that it sees the same number of 1s.

Note that this Lemma only describes a property of regular languages—if some language satisfies the Pumping Lemma that is *not* enough to conclude that said language is itself regular. Try to show that certain languages are not regular in Exercise 15.

That shows that regular languages are not powerful enough to describe the language of well-balanced brackets. Clearly that is a requirement when building any compiler, since the compiler needs to be able to treat nested statements. The reason for this is that a DFA cannot count, in the sense that there is no way it can remember how many opening brackets it has seen already, and how many of those have already been closed.

**Summary.** In this section we have shown that languages defined using regular expressions (the ‘regular languages’) are precisely the languages accepted by finite deterministic automata which are precisely the languages accepted by finite non-deterministic automata. While matching a word to a pattern is a recursive procedure, checking whether a word is accepted by an automaton is straightforward. Further we have seen that regular languages are closed under a number of operations, and that infinite regular languages have the property that words which are long enough will contain a substring which can be repeated arbitrarily many times. Regular languages are not powerful enough to capture most programming languages, but they are sufficient to treat certain sub-problems.

## 1.9 Exercises for Section 1

**Exercise 1** (a) Which of the following words match the given pattern?

Pattern	$a$	$ab$	$b$	$aba$	$abab$	$aab$	$aabb$	$aa$
$(ab)^*$								
$a^*b^*$								
$(a b)$								
$(a b)^*$								
$ab b a^*$								

(b) Give a regular expression for all the words which contain at least two letters  $a$  over the alphabet  $\{a, b\}$ .

(c) Give a regular expression for all the words over the alphabet  $\{a, b, c\}$  which do not contain the letter  $c$ .

**Exercise 2** (a) The Unix command `grep` allows you to take a pattern  $p$  and create a new pattern  $p+$ . This behaves almost like the Kleene star, but the pattern  $p$  must be matched at least once. Show that this does not give an increase in expressivity—in other words, this operator can be translated into a regular expression as given in Definition 4.

*[\*At first sight, the regular expressions found for example in the Unix or Linux version of the command `grep` appear to be more powerful—there are constructs that do not appear in our definition. Convince yourself that this makes no difference in practice.]*

(b) *(This one is hard to get exactly right. It is included here because it is about regular expressions rather than automata. You may want to postpone it and come back to it later.) Write down a regular expression which describes all those words which do not contain the substring 01. (It does not really matter what the alphabet is, although it should contain symbols other than 0 and 1 to cover the general case.) (Hint: Try to split words up into blocks to find the pattern that describes this.)*

**Exercise 3** (a) Design a DFA that recognizes all words over the alphabet  $\Sigma = \{0, 1, 2, 3\}$  that contain the substring 01.

(b) Design a DFA that recognizes all words over the alphabet  $\Sigma$  from above that do not contain the letter 3.

(c) Design a DFA that recognizes all words over the alphabet  $\Sigma$  from above which contain the substring 01 but which do not contain the substring 3.

(d) Find the regular expression describing all the words recognized by the DFA in Figure 18.

(e)\* *Consider how to formally define when a ‘relaxed DFA’ accepts a word, where a ‘relaxed DFA’ is presented by a graph of a relaxed standard informally discussed above, such as that in Figure 1 (as opposed to Figure 2).*

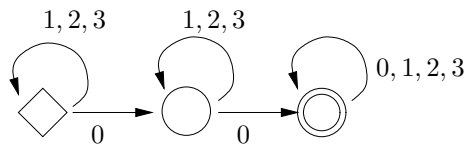


Figure 18: Which words does this DFA accept?

**Exercise 4** Design automata over the alphabet  $\{a, b, c\}$  recognizing the words matching the following patterns.

- (a)  $(a|b)cc$
- (b)  $cc(a|b)$
- (c)  $aa|bb|cc$
- (d)  $c(a|b)^*c$ .

Now assume that we want to accept all words which contain substrings matching the given patterns. What changes do you have to make?

**Exercise 5** Which words are accepted by the automaton in Figure 19? Give the corresponding regular expression.

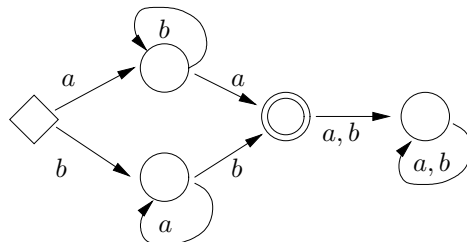


Figure 19: Which words does this DFA accept?

Finite automata can do a limited amount of counting.

**Exercise 6** (a) Design a DFA that accepts all words that have no more than five 0s. (Does it matter which alphabet is used?)

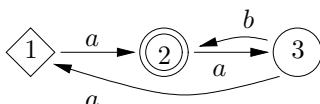
(b) Design a DFA that accepts all words with an even number of 0s.

(c) Design a DFA that accepts all words over  $\{0, 1\}$  which contain an even number of 0s and an odd number of 1s.

Sometimes we are interested in *not* accepting words matching a certain pattern.

**Exercise 7** Design a DFA that accepts all words which do not contain the substring 01 (compare Exercise 3 (a)).

**Exercise 8** Using the construction described in the proof of Proposition 1.1 find a pattern for the following automaton. Can you read off a pattern from the automaton? Does it match the one constructed?



**Exercise 9** Following the algorithm outlined before Proposition 1.3, construct an NFA with  $\tau$ -transitions that accepts all words that match the pattern  $(a|b)^*a(a|b)$ . Now do the same again, but only introduce  $\tau$ -transitions where they are needed.

**Exercise 10** Take the NFA constructed in Exercise 9 and remove the  $\tau$ -transitions.

**Exercise 11** Take the NFA from Exercises 9 and 10. How many states does the corresponding DFA have if we generate the full automaton described on page 18? Construct the DFA where only the reachable states are generated. Can you construct a DFA with even fewer states?

**Exercise 12** Construct a DFA for the regular expression  $((ab|a)^*$  following the steps outlined in the proof of Proposition 1.2. Feel free to leave out unnecessary  $\tau$ -transitions.

**Exercise 13** These are really, really hard. Don't spend much time on them.

(a)\* Describe a sequence of regular expressions of linearly increasing lengths such that the smallest recognizing automaton grows exponentially.

(b)\* Try the converse: Describe a sequence of finite automata for which the shortest regular expression grows exponentially in length.

**Exercise 14** (a) Given a DFA for a language  $L$ , construct a finite automaton for the language  $L^R$ .

(b) Given automata for languages  $L$  and  $L'$  construct one for the language  $L \cap L'$ . (Note that each character can only be read once, so putting one automaton after the other will not work.)

(c) Show that the language  $L$  over the alphabet  $\{a, b\}$  which consists of all words which contain the same number of occurrences of the substring  $ab$  as of the substring  $ba$  is regular.

**Exercise 15** (a) Use the Pumping Lemma to show that the set of all words containing more  $a$ s than  $b$ s is not regular

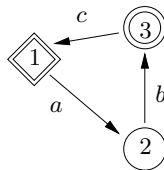
(b) Use the Pumping Lemma to show that the set of words consisting of well-balanced brackets is not regular.

## Assessed exercises, due in week 4

**Exercise 1.1** (a) Design a DFA that accepts precisely those words which are not accepted by the DFA in Figure 1 from the notes.

(b) Given a DFA  $A$  how can you define a DFA which accepts precisely those words which  $A$  does not accept? (Hint: You may find it easier to consider only automata that show all states, such as that in Figure 2.)

**Exercise 1.2** Use the algorithm given in the proof sketch of Proposition 1.1 to find a pattern which describes the same language as the automaton below. Make sure you simplify the expression wherever possible!



**Exercise 1.3** Give a non-deterministic automaton (with  $\tau$ -moves, if you like) that recognizes the language given by the pattern  $ab^*c^*|a^*b^*c$ . Then turn this automaton into a deterministic one by applying the algorithm from the notes. This will be easier if you simplify your automaton wherever possible.