COMP11212 Fundamentals of Computation
Part 1: Formal Languages

Andrea Schalk
`A.Schalk@manchester.ac.uk`

January 22, 2014

Organizational issues

When, what, where

This course will be taught as follows.

- Lectures: Will take place Wednesdays at 9.00 and Thursdays at 10.00 in Kilburn 1.1.
- Examples classes: Will take place from Week 2 as follows.

Groups	Time	Location
M+W	Mondays 1.00	IT407
B+X	Mondays 2.00	IT407
Y	Mondays 3.00	IT407
Z	Mondays 4.00	IT407

Two parts

This course consists of two distinct parts, taught by two different members of staff, Andrea Schalk and David Lester.

Lectures	From	To	Content	Taught by
1	29/01	29/01	Intro + start Part 1	David & Andrea
2–10	30/01	27/02	Part 1	Andrea
11–20	05/03	03/04	Part 2	David
21–22	30/09	01/05	Revision	Andrea & David

Examples classes in Weeks	belongs to
2–6	Part 1
7–11	Part 2

Assessment

There is *assessed coursework* in the form of exercises you are expected to prepare **before each examples class**, see below for more detail. The coursework is worth *25% of the overall mark for the course unit.*

The exam consists of four question, with two questions from each part of the course. Each question in the exam is worth twenty marks. Students have to answer *three out of the four questions.* This means your exam mark will be out of sixty. It will be worth *75% of your mark for this course unit.*

Coursework

You are expected to prepare exercises **before each examples class**, see the *exercise sheets at the end of the notes.* During the examples class these will be marked, you will get feedback on what you did, and you will also get help with the questions you could not do.

Each exercise is **marked out of two**, where the marks mean the following.

0: You did not make a serious attempt to solve the question.

1: You made a serious attempt but did not get close to finishing the question.

2: You did work your way through the exercise and made no serious mistakes.

There are four or five marked exercises for each examples class. Note that not all exercise are equally difficult, or require an equal amount of work. If you are seriously struggling with one exercise, move on to the next one (they don't usually build on each other). If you are struggling for time, try to do as many as you can, leaving out the more labour-intensive ones, but try to do something.

The rules for marking are as follows:

- To get a mark for any exercise you must produce all the work you have done for it *before the examples class* when a marker asks you to show it to them.
- The marker will ask to see *all your work* for an exercise, so please bring with you any rough work. It is possible to get one mark for an exercise where you haven't made much, or any, progress, provided your rough works give evidence that you have seriously tried it. I'd expect there to be at least a page or more of your trying things out to find an answer. If you bring just a final answer for an exercise that requires steps in between you will not get the marks for it.
- If you tried an exercise and got one mark for it, and you can substantially improve what you have done after a demonstrator has explained it to you, you may ask the demonstrator who marked you originally to remark that exercise, and improve your mark from 1 to 2. This is subject to *the demonstrator having time available in the examples class.*
- You **cannot** improve your mark from 0 to 2. Hence any exercise where you can only show work you have done during the examples class will not count.
- You may have at most one 'remarking' session per examples class (but this may include more than one exercise).

Note that the marker may ask you *how* you solved the various exercises before assigning you a mark for each of them. We also expect you to have all your rough work available—if you have only the solution where there is substantial work required to get there the demonstrator has been told to give you 0.

- If you cannot describe how you did the work then we have to assume that you plagiarized the solution and you will **get a mark of 0 for all your work that week**.

- If this happens twice **we will put a note in your file that you have been caught plagiarizing work**.

- Weeks in which you have done this will *not count towards the seven examples class threshold* (see below).

Note that your coursework marks **only** count if you have something recorded for at least **seven out of the ten examples classes**, or if there are mitigating circumstances.

Changes

Note that the course unit was taught very differently until (and including) 2011/2012. Here are the most important changes.

- There used to be three parts. The old second part (Part B in old exam papers) is no longer taught. Part 1 has been very slightly, and Part 2 (the old Part 3) substantially, expanded since then.

- In the old exams questions used to be worth ten marks each, with the exam being marked out of fifty.

- There used to be coursework *on top of* the exercises prepared for the examples classes. This no longer exists, but now the examples class prep is being marked.

About Part 1 of the course

As detailed on the previous page, Part 1 of this course unit is concerned with *formal languages* and is taught by Andrea Schalk. The following concerns organizational issues just for this first part.

Learning the material

The most important aims of this part of the course are to teach you

- a few concepts (such as formal language, regular expression, finite state automaton, context-free grammar) and
- how to solve a number of typical problems,

with an emphasis on the latter.

Exercises. The best way to learn how to solve problems is to do it, which is why the notes contain a great number of exercises. Those not covered in the exercise sheets provide further opportunities to practise, for example when revising for the exams. The exercises are part of the examinable material, although I won't ask you to recall any particular exercise in the exam. Some exercises have a *, which means that they are not part of the examinable material. Note that they are not necessarily difficult, although some of them are.

Lectures. I use the lectures to introduce the main ideas, and to explain the important algorithms, as well as suggesting some ways for solving typical problems.

Self-study. The course has only three contact hours a week, and you are expected to spend a substantial amount of time each week working on the material by studying the notes yourself, connecting the written material with what was introduced in the lectures, and solving exercises.

Notes. The notes are fairly substantial because they aim to cover the material in detail. Typically, for each new idea or algorithm, there is a formal description as well as one or two examples. For most people it is worthwhile trying to understand the example before tackling the general case, although in the notes the general case is sometimes described before the example (because it can be difficult to explain the idea using just a special case). Note that there is a glossary that you can use to quickly find the meaning of the various technical terms used.

Mathematical notation. The language used to describe much of theoretical computer science is that of mathematics, because that is the *only* way of describing with precision what we are talking about. In the main part of the notes I have tried to remain as informal as possible in my descriptions, although all the definitions given are formal. You will not be required to repeat these definitions in the exam, you only need to understand informally what they mean. There is an appendix which spells out in more detail how the language of mathematics is used to make rigorous the concepts and operations we use. If you only understand the informal side of the course you can pass the exam (and even get a very good mark) but to master the material (and get an excellent result in the exam) you will have to get to grips with the mathematical notation as well. A maximum of 10% of the final mark depends on the material in the Appendix.

Examples classes. The examples classes give you the opportunity to get help with any problems you have, either with solving exercises or with understanding the notes. There are four examples classes associated with this part of the course, in Weeks 2–5. For each of these you are expected to prepare by solving the key exercises on the appropriate sheet. It also suggests exercises to do if you are working through the Appendix, and there are suggestions for additional exercises you might want to do if you find the set work easy. The sheets can be found from pages 88 of the notes. We will check in each examples classes whether you have done the preparation, and the data will be entered into Arcade. **Solutions** for each exercise sheet will be made available on the webpage for this part of the course after the last associated examples class has taken place.

Marking criteria. The marking criteria for the assessed coursework are *stricter* than those for the exam. In particular, I ask you to follow various algorithms as described in the notes, whereas in the exam I'm happy if you can solve the problem at hand, and I don't mind how exactly you do that. In all cases it is important to show your work—if you only give an answer you may lose a substantial number of marks.

Revision. For revision purposes I suggest going over all the exercises again. Some of the exercises will probably be new to you since they are not part of the set preparation work for the examples classes. Also, you can turn all the NFAs you encounter along the way into DFAs (removing ϵ-transitions as required). Lastly, there are exams from previous years to practice on. If you can do the exercises you will do well in the exam. You will not be asked to repeat definitions, but knowing about properties of regular and context-free languages may be advantageous. You should also be aware of the few theorems, although you will not be asked to recite them.

Webpage for the course. `http://studentnet.cs.manchester.ac.uk/ugt/COMP11212/`

Reading

For this part of the course these notes cover all the examinable material. However, sometimes it can be useful to have an additional source to see the same material introduced and explained in a slightly different way. Also, there are new examples and exercises to be looked at. For this purpose you may find one of the following useful.

M. Sipser. **Introduction to the Theory of Computation.** *PWS Publishing Company*, 1997. ISBN 0-534-94728-X.

This is a fairly mathematical book that nonetheless tries to keep mathematical notation at a minimum. It contains many illustrated examples and aims to explain ideas rather than going through proofs mechanically. A 2005 edition is also available. Around £ 50—this book is very well thought of and even used copies are quite expensive. Relevant to this course: Chapters 0 (what hasn't yet been covered by COMP10020), 1 and 2.1

J.E. Hopcroft, R. Motwani, and J.D. Ullman. **Introduction to Automata Theory, Languages, and Computation.** *Addison Wesley*, second edition, 2001. ISBN 0-201-44124-1.

This account is derived from the classical textbook on the subject. It is more verbose than Sipser, and it aims to include up-to-date examples and applications, and it develops the material in much detail. A number of illustrated examples are also included. I have heard that this book is very popular among students. 2006/7 editions are also available. About £50. Relevant for this course are Chapters 1 to 5.

Get in touch

Feedback. I'm always keen to get feedback regarding my teaching. Although the course has been taught a few times the notes probably still contain some errors. I'd like to hear about any errors that you may find so that I can correct them. I will make those available at `http://studentnet.cs.manchester.ac.uk/ugt/COMP11212/` and fix them for future years. You can talk to me after lectures, or send me email (use the address on the title page).

Reward! If you can find a substantial mistake in the lecture notes (that is, more than just a typo, or a minor language mistake) you get a chocolate bar.

Wanted. I'm also on the lookout for good examples to use, either in the notes or in the lectures. These should be interesting, touch important applications of the material, or be fun. For a really good reasonably substantial such example the reward is a chocolate bar.

Acknowledgements. I would like to thank Howard Barringer, Pete Jinks, Djihed Afifi, Francisco Lobo, Andy Ellyard, Ian Worthington, Peter Sutton, James Bedford, Matt Kelly, Cong Jiang, Mohammed Sabbar, Jonas Lorenz, Tomas Markevicius and Joe Razavi for taking the time to help me improve these notes.

Contents

Chapter 1

What it is all about

Computers need to be able to interpret input from a keyboard. They have to find commands and then carry out appropriate actions. With a command line interface this may appear a fairly simple problem, but, of course, the keyboard is not the only way of feeding a computer with instructions. Computers also have to be able to read and interpret files, for example when it comes to compiling a program from a language such as Java, C or ML and they then have to be able to run the resulting code. In order to do this computers have to be able to split the input into strings, parse these strings, and turn those into instructions (such as Java bytecode) that the machine can carry out.

How does a computer organize potentially quite large input? How does it decide what variables to create, and what values to give these? In this part of *Fundamentals of Computation* we look at some techniques used for this purpose.

In order to organize (or parse) the given text-like input a computer has to be able to recognize specific strings, for example simple commands (such as `if` or `else`). The ability to find certain strings is useful in other contexts as well. When a search engine such as Google looks for a string or phrase entered by a user it certainly has to be capable of telling when it has found the string in question. However, most search engines do their job in a much more clever way than that: They will also recognize plurals of strings entered, and if such a string should be a verb of the English language it will typically recognize different forms, such as *type*, *typing* and *typed.* How can we come up with clever ways of searching for something more than just a given fixed string? Again this is an issue that this course covers.

How does the web server for the Intranet in the School of Computer Science check whether you are accessing it from within the University?

How do online shopping assistants work? They find a given product at various shops and tell you what it costs. How do they extract the information, in particular since some of the pages describing the product are created on demand only?

Imagine you had to write a piece of code that accepts a string as an input and checks whether this string could be a valid email address.

Imagine you had to write a program that goes through a number of emails in a folder and returns all the subject lines for those (every email client has a way of doing this, of course). How would you do that?

Imagine you had to write a program that goes through a bunch of documentation files and checks whether they contain any double typed words (such as ‘the the’). This is a very common mistake, and a professional organization would want to remove these before delivering the software to a customer.

What if

- Your program had to work across lines (and an arbitrary number of spaces)?
- You had to take into account that the first doubled word might be capitalized?
- The two doubled words might be separated by html tags?

Imagine you had to write code for a vending machine that takes coins and in return delivers variously priced goods. How does the machine keep track of how much money the user has put in so far, and how to produce correct change from the coins it holds? It's your job to keep track of all the cases that may arise.

Imagine you had to write a piece of code that takes as input a file containing a Java program and checks whether the curly brackets { } are all properly balanced.

This course is going to help you with understanding how such tasks can be carried out. However, we won't spend much time writing code. The tools we describe here could be used when using the Unix `grep` command, or when programming in languages such as Perl, Python, Tcl, or even when using the GNU Emacs editor. It is more important to me that you understand the ideas, and how to express them in a reasonably formal way (so that programming them then becomes easy).

The material can be split into three parts, dealing with the following issues.

- How do we describe to a computer what strings to look for?
- How can we come up with the right description to solve our problem?
- How can we write code that checks whether some other piece of code is correctly formed, whether the opening and closing brackets match, or whether a webpage is written in valid html?

Chapter 2

Describing languages to a computer

In order to solve the kinds of problems that are mentioned in Chapter 1 we need to be able to describe to a computer what it is looking for.

Only in very simple cases will this consist of just one or two strings—in general, we want the computer to look for a much larger collection of words. This could be the set of all possible IP addresses within the University, or it could be the collection of all strings of the form 'Subject:...' (to pick out emails on a particular topic), or it could be the set of all strings of the form $s␣s$ (in the simplest case of finding all doubled words—this one won't take care of doubled words spread over two lines, nor of the first word being capitalized).

2.1 Terminology

In order to talk about how we do this in practice we have to introduce a few terms so that we can talk about the various concepts involved. You can find formal definitions of these Appendix A.

- A **symbol** is a building block for a string. Symbols cannot be subdivided, they are the atoms of everything we build. In the theory of formal languages they are usually called **letters**. Examples: a, A, 0, 1, %, @. We use letters from the end of the Roman alphabet, such as x, y, z to refer to an arbitrary symbol.
- An **alphabet** is a collection of letters. We think of it as a set. At any given time we are only interested in those words we can build from a previously specified alphabet. Examples: $\{0, 1\}$, $\{0, 1, \ldots, 9\}$, $\{a, b, \ldots, z\}$, $\{a, b, \ldots, z, A, B, \ldots, Z\}$,. We use capital Greek letters, typically Σ, to refer to an arbitrary alphabet.
- A **string** is something we build from 0 or more symbols. In the theory of formal languages we usually speak of **words**. Examples: $ababab$, 0, 1001. Note that every letter can be viewed as a one-letter word. We use letters s, t, u to refer to an arbitrary word.
- The **empty word** (which consists of 0 letters) is a bit difficult to denote. We use ϵ (the Greek letter 'ϵ') for this purpose.
- **Concatenation** is the operation we perform on words (or letters) to obtain longer words. When concatenating a with b we get ab, and we

can concatenate that with the word ba to obtain $abba$. If we concatenate 0 letters we get the word ϵ. When we concatenate any word with the word ϵ we obtain the same word.

- We use the notation of powers for concatenation as follows: If s is a word then $(s)^n$ is the word we get by concatenating n copies of s. For example, $(010)^3 = 010010010$, $1^2 = 11$, $b^0 = \epsilon$ and $c^1 = c$.
- A **language** is a collection of words, which we think of as a set. Examples are $\{\epsilon\}$, $\emptyset$, $\{ab, abc, aba\}$ and $\{a^n \mid n \in \mathbb{N}\}$. We use letters such as $\mathcal{L}$, $\mathcal{L}_1$ and $\mathcal{L}'$ to refer to an arbitrary language.

In these notes we are interested in how we can describe languages in various ways.

If a language is finite then we can describe it quite easily: We just have to list all its elements. However, this method fails when the language is infinite, and even when the language in question is very large. If we want to communicate to a computer that it is to find all words of such a language we have to find a concise description.

2.2 Defining new languages from old ones

One way of describing languages is using set-theoretic notation (see Appendix A for more detail). In the main development here we try to avoid being overly mathematical, but there are some operations on languages we need to consider. There are seven of them:

- **Union.** Since languages are just sets we can form their unions.
- **Intersection.** Since languages are merely sets we can form their intersections.
- **Set difference.** If $\mathcal{L}_1$ and $\mathcal{L}_2$ are languages we can form

$$\mathcal{L}_1 \setminus \mathcal{L}_2 = \{s \in \mathcal{L}_1 \mid s \notin \mathcal{L}_2\}.$$

- **Complement.** If $\mathcal{L}$ is the language of all words over some alphabet Σ, and $\mathcal{L}'$ is a subset of $\mathcal{L}$ then the complement of $\mathcal{L}'$ in $\mathcal{L}$ is $\mathcal{L} \setminus \mathcal{L}'$, the set of all words over Σ which are not contained in $\mathcal{L}'$.
- **Concatenation.** Because we can concatenate words we can use the concatenation operation to define new languages from existing ones by extending this operation to apply to languages as follows. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be languages over some alphabet Σ. Then

$$\mathcal{L}_1 \cdot \mathcal{L}_2 = \{s \cdot t \mid s \in \mathcal{L}_1 \text{ and } t \in \mathcal{L}_2\}.$$

- **n-ary Concatenation.** If we apply concatenation to the same language by forming $\mathcal{L} \cdot \mathcal{L}$ there is no reason to stop after just one concatenation. For an arbitrary language $\mathcal{L}$ we define

$$\mathcal{L}^n = \{s_1 \cdot s_2 \cdot \cdots \cdot s_n \mid s_i \in \mathcal{L} \text{ for all } 1 \leq i \leq n\}.$$

We look at the special case

$$\mathcal{L}^0 = \{s_1 \cdots s_n \mid s_i \in \mathcal{L} \text{ for all } 1 \leq i \leq n\} = \{\epsilon\},$$

since ϵ is what we get when concatenating 0 times.

- **Kleene star.** Sometimes we want to allow any (finite) number of concatenations of words from some language. The operation that does this is known as the **Kleene star** and written as $(-)^*$. The definition is:

  $$\begin{aligned}\mathcal{L}^* &= \{s_1 s_2 \cdots s_n \mid n \in \mathbb{N}, s_1, s_2, \ldots, s_n \in \mathcal{L}\}\\ &= \bigcup_{n \in \mathbb{N}} \mathcal{L}^n.\end{aligned}$$

  Note that

  $$\emptyset^* = \bigcup_{n \in \mathbb{N}} \emptyset^n = \{\epsilon\} = \mathcal{L}^0$$

  for all $\mathcal{L}$, which may be a bit unexpected.

Exercise 1. To gain a better understanding of the operation of concatenation on languages, carry out the following tasks.

(a) Calculate $\{a, aaa, aaaaaa\} \cdot \{\epsilon, bb, bbb\}$.

(b) Calculate $\{\epsilon, a, aa\} \cdot \{aa, aaa\}$.

(c) Calculate $\{a, a^3, a^6\} \cdot \{b^0, b^2, b^3\}$.

(d) Describe the language $\{0^m 1^n \mid m, n \in \mathbb{N}\}$ as the concatenation of two languages.

(e) Calculate $\{0, 01, 001\}^2$.

Exercise 2. To gain a better understanding of the Kleene star operation carry out the following tasks. (You may use any way you like to describe the infinite languages involved: Just using plain English is the easiest option. I would prefer not to see ... in the description.)

(a) Calculate $\{a, b\}^*$, $\{a\}^* \cup \{b\}^*$, $\{a\}^* \cap \{b\}^*$, $\{aa\}^* \setminus \{aaaa\}^*$ and the complement of $\{a\}^*$ in the language of all words over the alphabet $\{a, b\}$.

(b) Calculate $\{0^{2n} \mid n \in \mathbb{N}\}^*$.

(c) Describe the set of all words built from a given alphabet Σ using the Kleene star operation.

2.3 Describing languages through patterns

By definition a languages is a set. While sets are something mathematicians can deal with very well, and something that computer scientists have to be able to cope with, computers aren't at all well suited to make sense of expressions such as

$$\{(01)^n \mid n \in \mathbb{N}\}.$$

There are better ways of showing a computer what all the words are that we mean in a particular case. We have already seen that we can express the language $\mathcal{L}$ in a different way, namely as

$$\{01\}^*.$$

If there is only one word in a language, here $\{01\}$, we might as well leave out the curly brackets and write[1] $(01)^*$. That is actually something a

[1] Can you see why we need brackets here?

machine can cope with, although we would have to use a slightly different alphabet, say `(01)^*`. All a computer now has to do is to compare: Does the first character[2] of my string equal 0? Does the next one[3] equal 1? And so on.

What we have done here is to create a *pattern.* It consists of various characters of the alphabet, concatenated. We are allowed to apply the Kleene star to any part of the string so created, and we use brackets ( ) to indicate which part should be affected by the star. A computer can then *match* this pattern.

Are these all the patterns we need? Not quite. How, for example, would we describe the language

$$\{0^n \mid n \in \mathbb{N}\} \cup \{1^n \mid n \in \mathbb{N}\} = \{x^n \mid x = 0 \text{ or } x = 1\}?$$

We can't use either of 0^*1^* or (arguably worse) $(01)^*$ because both of these include words that contain 0 as well as 1, whereas any word in our target language consists entirely of 0s or entirely of 1s. We need to have a way of saying that either of two possibilities might hold. For this, we use the symbol $|$. Then we can use $0^*|1^*$ to describe the language above.

Exercise 3. Which of the following words match the given patterns?

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

Exercise 4. Describe all the words matching the following patterns. For finite languages just list the elements, for infinite ones, try to describe the words in question using English. If you want to practise using set-theoretic notation, add a description in that format.

(a) $(0|1|2)$

(b) $(0|1)(0|2)2$

(c) $(01|10)$

(d) 0^*1,

(e) $(01)^*1$,

(f) 0^*1^*,

(g) $(010)^*$,

(h) $(01)^*0^*$,

(i) $(01)^*(01)^*$.

(j) $(0|1)^*$

(k) $0^*|1^*|2^*$

(l) $0|1^*|2$

[2] What should the computer do if the string is empty?

[3] What if there isn't a next symbol?

2.4 Regular expressions

So far we have been using the idea of a pattern intuitively—we have not said how exactly we can form patterns, nor have we properly defined when a word matches a pattern. It is time to become rigorous about these issues.

For reason of completeness we will need two patterns which seem a bit weird, namely $\boldsymbol{\epsilon}$ (the pattern which is matched precisely by the empty word ϵ) and $\boldsymbol{\emptyset}$ (the pattern which is matched by no word at all).

Definition 1. *Let* Σ *be an alphabet. A* ***pattern*** *or* **regular expression over** Σ *is any word over*

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

generated by the following inductive definition.

Empty pattern *The character* $\boldsymbol{\emptyset}$ *is a pattern;*

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

Letters *every letter from* Σ *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* $(p*)$*.*

In other words we have defined a *language*[4], namely the language of all regular expressions, or patterns. Note that while we are interested in words over the alphabet Σ we need *additional symbols* to create our patterns. That is why we have to extend the alphabet to Σ^{pat}.

In practice we will often leave out some of the brackets that appear in the formal definition—but only those brackets that can be uniquely reconstructed. Otherwise we would have to write ((0|1)*0) instead of the simpler (0|1)*0. In order to be able to do that we have to define how to put the brackets back into such an expression. We first put brackets around any occurrence of $*$ with the sub-pattern immediately to its left, then around any occurrence of concatenation, and lastly around the alternative operator |. Note that every regular expression with all its brackets has precisely one way of building it from the rules—we say that patterns are *uniquely parsed.* This isn't quite true once we have removed the brackets: (0|(1|2)) turns into 0|1|2 as does ((0|1)|2). However, given the way we use regular expressions this does not cause any problems.

Note that many computer languages that use regular expressions have additional operators for these (see Exercise 7). However, these exist only for the convenience of the programmer and don't actually make these regular expressions more powerful. We say that they have the same *power of expressivity.* Whenever I ask you to create a regular expression or pattern it is Definition 1 I expect you to follow.

[4] In Chapter 4 we look at how to describe a language like that—it cannot be done using a pattern.

2.5 Matching a regular expression

This language is **recursively defined**: There are base cases consisting of the $\boldsymbol{\emptyset}$, $\boldsymbol{\epsilon}$ and all the letters of the underlying alphabet, and three ways of constructing new patterns from old ones. If we have to define something for all patterns we may now do so by doing the following:

- Say how the definition works for each of the base cases;
- assuming we have defined the concept for p, p_1 and p_2, say how to obtain a definition for
  - p_1p_2,
  - $p_1|p_2$ and
  - p^*.

This is an example of a recursive definition, and you will see more of these in COMP10020.[5]

Note that in the definition of a pattern no meaning is attached to any of the operators, or even the symbols that appear in the base cases. We only find out what the intended meaning of these is when we define when a word matches a pattern. We do this by giving a recursive definition as outlined above.

Definition 2. *Let p be a pattern over an alphabet Σ and let s be a word over Σ. We say that s* ***matchinges*** *p if one of the following cases holds:*

Empty word *The empty word ϵ matches the pattern $\boldsymbol{\epsilon}$;*

Base case *the pattern $p = x$ for a character x from Σ and $s = x$;*

Concatenation *the pattern p is a concatenation $p = (p_1p_2)$ and there are words s_1 and s_2 such that s_1 matches p_1, s_2 matches p_2 and s is the concatenation of s_1 and s_2;*

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

Kleene star *the pattern p is of the form $p = (q*)$ and s can be written as a finite concatenation $s = s_1s_2 \cdots s_n$ such that s_1, s_2, $\ldots$, s_n all match q; this includes the case where s is empty (and thus an empty concatenation, with $n = 0$).*

Note that there is no word matching the pattern $\boldsymbol{\emptyset}$.

Exercise 5. Calculate all the words matching the patterns $\boldsymbol{\epsilon}$ and a (for $a \in \Sigma$) respectively.

Exercise 6. For the given pattern, and the given word, employ the recursive Definition 2 to demonstrate that the word does indeed match the pattern:

(a) the pattern $(ab)^*a$ and the word aba.

(b) the pattern $(0|1)^*10$ and the word 10010,

(c) the pattern $(0^*|1^*)10$ and the word 0010,

[5] Students who are on the joint honours CS and Maths programme don't take COMP10020, but they should be able to grasp these ideas without problems.

(d) the pattern $(abc)^*a$ and the word a.

Exercise 7. Here are some examples of the usage of regular expressions in 'the real world'.

(a) Print out the manual page for the Unix command `grep` (type `man grep` to get that page). Now argue that for every regular expression understood by `grep` there is a regular expression according to Definition 1 that has precisely the same words matching it.[6]

(b) Give a command-line instruction which will show all the subjects and authors of mails contained in some directory. Make sure to test your suggested answer on a computer running Linux. *Hint: Try using* `egrep`.

(c) Give a regular expression that matches all IP addresses owned by the University of Manchester. *You probably want to use some kind of shortcut notation.*

2.6 The language described by a regular expression

Given a pattern we can now define a language based on it.

Definition 3. *Let p be a regular expression over an alphabet Σ. The* **language defined by pattern** *p, $\mathcal{L}(p)$ is the set of all words over Σ that match p. In other words*

$$\mathcal{L}(p) = \{s \in \Sigma^* \mid s \textit{ matches } p\}.$$

Note that different patterns may define the same language, for example $\mathcal{L}(0^*) = \mathcal{L}(\boldsymbol{\epsilon}|00^*)$.

Note that we can also define the language given by a pattern in a different way, namely recursively.

- $\mathcal{L}(\boldsymbol{\emptyset}) = \boldsymbol{\emptyset}$;
- $\mathcal{L}(\boldsymbol{\epsilon}) = \{\epsilon\}$;
- $\mathcal{L}(x) = \{x\}$ for $x \in \Sigma$;
- and for the operations
  - $\mathcal{L}(p_1p_2) = \mathcal{L}(p_1) \cdot \mathcal{L}(p_2)$;
  - $\mathcal{L}(p_1|p_2) = \mathcal{L}(p_1) \cup \mathcal{L}(p_2)$;
  - $\mathcal{L}(p^*) = (\mathcal{L}(p))^*$.

We can use this description to calculate the language defined by a pattern as in the following example.

$$\begin{aligned}\mathcal{L}((0|1)^*) &= (\mathcal{L}(0|1))^* \\ &= (\mathcal{L}(0) \cup \mathcal{L}(1))^* \\ &= (\{0\} \cup \{1\})^* \\ &= \{0,1\}^*\end{aligned}$$

This is the language of all words over the alphabet $\{0, 1\}$.

[6] Theoreticians want their patterns with as few cases as possible so as to have fewer cases for proofs by induction. Practicians want lots of pre-defined shortcuts for ease of use. This exercise shows that it doesn't really matter which version you use.

Exercise 8. Use this recursive definition to find the languages defined by the following patterns, that is the languages $\mathcal{L}(p)$ for the following p.

(a) $(0^*|1^*)$

(b) $(01)^*0$

(c) $(00)^*$

(d) $((0|1)(0|1))^*$.

Can you describe these languages in English?

In order to tell a computer that we want it to look for words belonging to a particular language we have to find a pattern that describes it. The following exercise lets you practise this skill.

Exercise 9. Find[7] a regular expression p over the alphabet $\{0, 1\}$ such that the language defined by p is the one given. *Hint: For some of the exercises it may help not to think of a pattern that is somehow formed like the strings you want to capture, but to realize that as long as there's one way of matching the pattern for such a string, that's good enough.*

(a) All words that begin with 0 and end with 1.

(b) All words that contain at least two 0s.

(c) All words that contain at least one 0 and at least one 1.

(d) All words that have length at least 2 and whose last but one symbol is 0.

(e) All words which contain the string 11, that is, two consecutive 1s.

(f) All words whose length is at least 3.

(g) All words whose length is at most 4.

(h) All words that start with 0 and have odd length.

(i) All words for which every letter at an even position is 0.

(j) All words that aren't equal to the empty word.

(k) All words that contain at least two 0s and at most one 1.

(l) All words that aren't equal to 11 or 111.

(m) All words that contain an even number of 0.

(n) All words whose number of 0s is divisible by 3.

(o) All words that do not contain the string 11.

(p) All words that do not contain the string 10.

(q) All words that do not contain the string 101.

(r)* All words that contain an even number of 0s and whose number of 1s is divisible by 3.

[7] If you find the last few of these really hard then skip them for now. The tools of the next chapter should help you finish them.

Exercise 10. Find a regular expression p over the alphabet $\{a, b, c\}$ such that the language defined by p is the one given.

(a) All the words that don't contain the letter c.

(b) All the words where every a is immediately followed by b.

(c) All the words that do not contain the string ab.

(d) All the words that do not contain the string aba.

2.7 Regular languages

We have therefore defined a certain category of languages, namely those that can be define using a regular expression. It turns out that these languages are quite important so we give them a name.

Definition 4. *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 $\mathcal{L} = \mathcal{L}(p)$.*

Regular languages are not rare, and there are ways of building them. Nonetheless in Chapter 4 we see that not all languages of interest are regular.

Proposition 2.1. *Assume that $\mathcal{L}$, $\mathcal{L}_1$, and $\mathcal{L}_2$ are regular languages over an alphabet Σ. Then the following also are regular languages.*

(a) $\mathcal{L}_1 \cup \mathcal{L}_2$

(b) $\mathcal{L}_1 \cdot \mathcal{L}_2$

(c) $\mathcal{L}^n$

(d) $\mathcal{L}^*$

Regular expressions are very handy when it comes to communicating to a computer quite complicated collections of words that we might be looking for. However, coming up with a regular expression that describes precisely those words we have in mind isn't always easy.

In the next chapter we look at different ways of describing languages—ways which make it easier for human beings to ensure that they are describing the language they had in mind. There is then an algorithm that translates one of these descriptions into a pattern.

2.8 Summary

- In order to describe a language to a computer we can use *regular expressions*, also known as *patterns.*
- For regular expressions we have the notion of it being *matched* by some particular word.
- Each regular expression defines a language, namely the set of all words that match it.
- We say that a language is *regular* if there is a pattern that defines it.

Chapter 3

How do we come up with patterns?

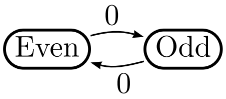
As you should have noticed by now (if you have done the last two exercises from the previous section) coming up with a pattern that describes a particular language can be quite difficult. One has to develop an intuition about how to think of the words in the desired language, and turn that into a characteristic for which a pattern can be created. While regular expressions give us a format that computers understand well, human beings do much better with other ways of addressing these issues.

3.1 Using pictures

Imagine somebody asked you to check a word of an unknown length to see whether it contains an even number of 0s. The word is produced one letter at a time. How would you go about that?

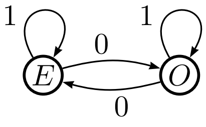
In all likelihood you would not bother to count the number of 0s, but instead you would want to remember whether you have so far seen an odd, or an even, number of 0s. Like flicking a switch, when you see the first 0, you'd remember 'odd', when you see the second, you'd switch back (since that's the state you'd start with) to 'even', and so forth.

If one wanted to produce a description of this idea it might look like this:

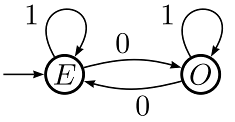


Every time we see 0 we switch from the 'even' state to the 'odd' state and *vice versa*.

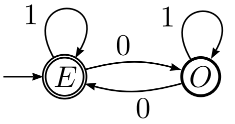
If we wanted to give this as a description to somebody else then maybe we should also say what we are doing when we see a letter other than 0, namely stay in whatever state we're in. Let's assume we are talking about words consisting of 0s and 1s. Also, we'd like to use circles for our states because they look nicer, so we'll abbreviate their names.



So now somebody else using our picture would know what to do if the next letter is 0, and what to do if it is 1. But how would somebody else know where to begin?



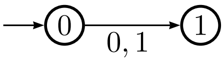
We give them a little arrow that points at the state one should start in. However, they would still only know whether they finished in the state called E or the one called O, which wouldn't tell them whether this was the desired outcome or not. Hence we mark the state we want to be in when the word comes to an end and now we do have a complete description of our task.



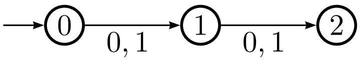
We use a double circle for a state to indicate that if we end up there then the word we looked at satisfied our criteria.

Let's try this idea on a different problem. Let's assume we're interested in whether our word has 0 in every position that is a multiple of three, that is, the third, sixth, ninth, *etc*, letters, if they exist, are 0.

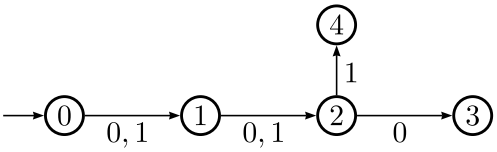
So we start in some state, and we don't care whether the first letter is 0 or 1, but we must remember that we've seen one letter so that we can tell when we have reached the first letter, so we'd draw something like this:



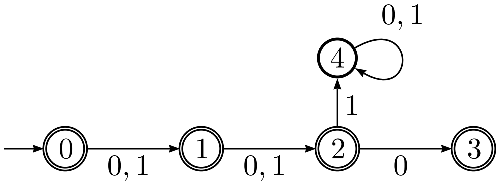
Similarly for the second letter.



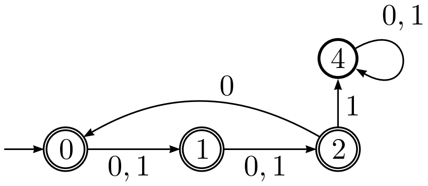
Now the third. This time something happens: If we see 0, we're still okay, but if we see 1 then we need to reject the word.



So what now? Well, if we reach the state labelled 4 then no matter what we see next, we will never accept the word in question of satisfying our condition. So we simply think of this state as one that means we won't accept a word that ends there. All the other states are fine—if we stop in any of them when reading a word it does satisfy our condition. So we mark all the other states as good ones to end in, and add the transitions that keep us in state 4.



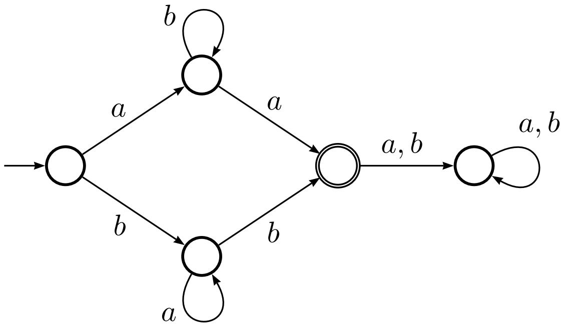
But what if the word is okay until state 3? Then we have to start all over again, not caring about the next letter or the one after, but requiring the third one to be 0. In other words, we're in the same position as at the start—so the easiest thing to do is *not to create state* 3, but instead to have that edge go back to state 0.



Exercise 11. For the languages described in parts (b) and (c) of Exercise 9, draw a picture as in the examples just given.

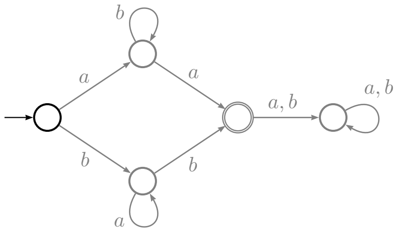
3.2 Following a word

Consider the following picture.

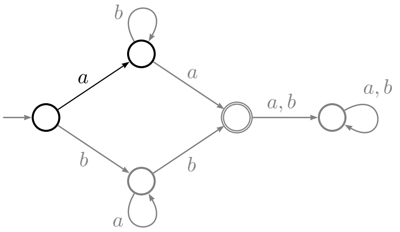


What happens when we try to follow a word, say *abaa*?

To begin with we are in the start state.

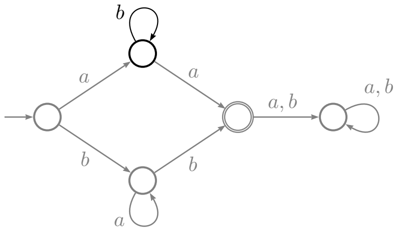


We see *a*, so we follow the edge labelled *a*. *abaa*.

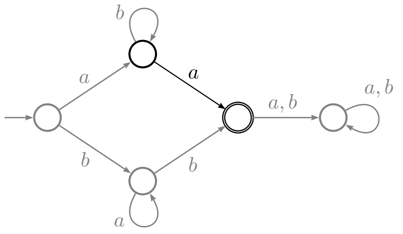


Having followed the first letter, we can drop it.

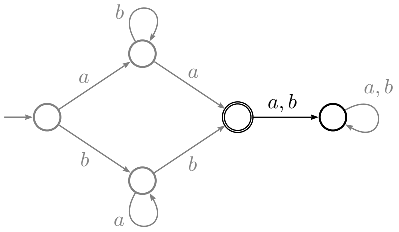
Now we see the letter b, so we go once round the loop in our current state. baa.



Now we have aa so we follow the edge labelled a



Lastly we have the word a, so we follow the edge to the right, labelled a, b.



We end up in a non-accepting state so this wasn't one of the words we were looking for.

Exercise 12. Follow these words through the above automaton. If you can, try to describe the words which end up in the one desirable state (the double-ringed one).

(a) *abba*

(b) *baba*

(c) *baab*.

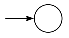
3.3 Finite state automata

So far we have been very informal with our pictures. It is impossible to define an automaton without becoming fairly mathematical. For any of these pictures we must know which *alphabet* Σ we are considering.

Every automaton consists of

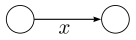
- a number of states,

- one state that is *initial*,



- a number (possibly 0) of states that are *accepting*

- for every state and every letter x from the alphabet Σ precisely one *transition* from that state to another labelled with x.



Formally we may consider the set of all the states in the automaton, say Q. One of these states is the one we want to start in, called the *start state* or the *initial state*. Some of the states are the ones that tell us if we end up there we have found the kind of word we were looking for. We call these *accepting states*. They form a subset, say F, of Q.

The edges in the graph are a nice way of visualizing the transitions. Formally what we need is a function that

- takes as its input
  - a state and
  - a letter from Σ

  and returns
- a state.

We call this the *transition function*, δ. It takes as inputs a state and a letter, so the input is a pair (q, x), where $q \in Q$ and $x \in \Sigma$. That means that the input comes from the set

$$Q \times \Sigma = \{(q, x) \mid q \in Q, x \in \Sigma\}.$$

Its output is a state, that is an element of Q. So we have that

$$\delta : Q \times \Sigma \longrightarrow Q.$$

The formal definition then is as follows.

Definition 5. *Let Σ be a finite alphabet of symbols. A* ***(deterministic) finite***[1] ***automaton****, short* ***DFA****, over Σ 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 δ *which for every state $q \in Q$ and every symbol $x \in \Sigma$ returns the next state $\delta(q, x) \in Q$, so δ is a function from $Q \times \Sigma$ to Q. When $\delta(q, x) = q'$ we often write*

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

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

Sometimes people also refer to a **finite state machine**.

Note that for every particular word there is precisely *one path* through the automaton: We start in the start state, and then read off the letters one by one. The transition function makes sure that we will have precisely one edge to follow for each letter. When we have followed the last letter of the word we can read off whether we want to accept it (if we are in an accepting state) or not (otherwise). That's why these automata are called *deterministic*; we see non-deterministic automata below.

For every word $x_1 x_2 \cdots x_n$ we have a uniquely determined sequence of states $q_\bullet, q_1, \ldots, q_n$ such that

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

We *accept* the word if and only if the last state reached, q_n is an accepting state. Formally it is easiest to define this condition as follows.

[1] All our automata have a finite number of states and we often drop the word 'finite' when referring to them in these notes.

Definition 6. *A* ***word*** $s = x_1 \cdots x_n$ ***over*** Σ ***is accepted by the deterministic finite automaton*** $(Q, q_\bullet, F, \delta)$ *if*

- $\delta(q_\bullet, x_1) = q_1$,
- $\delta(q_1, x_2) = q_2$,
- $\ldots$,
- $\delta(q_{n-1}, x_n) = q_n$ *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.

Just as was the case for regular expression we can view a DFA as defining a language.

Definition 7. *We say that a* ***language is recognized by a finite automaton*** *if it is the set of all words accepted by the automaton.*

Exercise 13. Design DFAs that accept precisely those words given by the languages described in the various parts of Exercise 9 (you should already have taken care of (b) and (c)). Note that (r) is no longer marked with $*$ here.

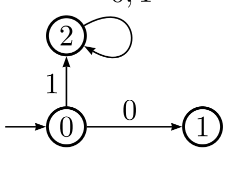
Exercise 14. Do the same for the languages described in Exercise 10.

3.4 Non-deterministic automata

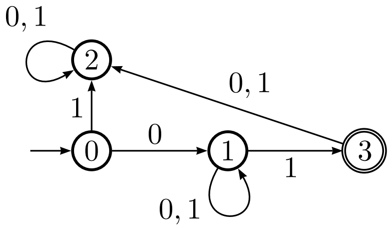
Sometimes it can be easier to draw an automaton that is *non-deterministic.* What this means is that from some state, there may be several edges labelled with the *same letter.* As a result, there is then no longer a unique path when we follow a particular word through the automaton. Hence the procedure of following a word through an automaton becomes more complicated—we have to consider a number of possible paths we might take.

When trying to draw an automaton recognizing the language from Exercise 9 (a) you may have been tempted to draw such an automaton.

The language in question is that of all words over $\{0, 1\}$ that begin with 0 and and with 1. We know that any word we accept has to start with 0, and if the first letter we see is 1 then we never accept the word, so we begin by drawing something like this.



Now we don't care what happens until we reach the last symbol, and when that is 1 we want to accept the word. (If it wasn't the last letter then we shouldn't accept the word.) The following would do that job:



But now when we are in state 1 and see 1 there are two edges we might follow: The loop that leads again to state 1 or the edge that leads to state 3. So now when we follow the word 011 through the automaton there are two possible paths:

From state 0, read 0, go to state 1.	From state 0, read 0, go to state 1.
Read 1 and go to state 3.	Read 1 and go to state 1.
Read 1 and go to state 2.	Read 1 and go to state 3.
Finish not accepting the word.	Finish accepting the word.

We say that the automaton accepts the word if there is *at least one such path* that ends in an accepting state.

So how does the definition of a non-deterministic automaton differ from that of a deterministic one? We still have a set of states Q, a particular start state $q_\bullet$ in Q, and a set of accepting states $F \subseteq Q$.

However, it no longer is the case that for every state and every letter from x there is precisely one edge labelled with x, there may be several. What we no longer have is a transition *function*. Instead we have a transition *relation*.[2] Given a state q, a letter x and another state q' the relation δ tells us whether or not there is an edge labelled x from q to q'.

Exercise 15. Go back and check your solutions to Exercises 13 and 14. Were they all deterministic as required? If not, redo them.

We can turn this idea into a formal definition.

Definition 8. *A* ***non-deterministic finite automaton****, short* ***NFA****, is given by*

- *a finite non-empty set Q of states,*
- *a start state $q_\bullet$ in Q,*
- *a subset F of Q of accepting states as well as*
- *a* transition relation *δ which relates a pair consisting of a state and a letter to a state. We often write*

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

if (q, x) is δ-related to q'.

We can now also say when an NFA accepts a word.

[2] You will meet relations also in COMP10020.

Definition 9. *A word $s = x_1 \cdots x_n$ over Σ* ***is accepted by the non-deterministic finite automaton*** *$(Q, q_\bullet, F, \delta)$ if there are states*

$$q_0 = q_\bullet, q_1, \dots, q_n$$

such that for all $0 \leq i < n$, δ relates (q_i, x_i) to q_{i+1} and such that $q_n \in F$, that is, q_n is an accepting state. The ***language recognized by an NFA*** *is the set of all words it accepts.*

An NFA therefore accepts a word $x_1 x_2 \cdots x_n$ if there are states

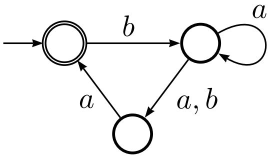
$$q_\bullet = q_0, q_1, \dots, q_n$$

such that

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

and q_n is an accepting state. However, the sequence of states is no longer uniquely determined, and there could potentially be many.

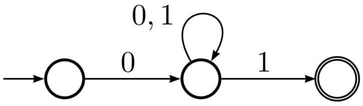
Exercise 16. Consider the following NFA.



Which of the following words are accepted by the automaton? ϵ, *aba*, *bbb*, *bba*, *baa*, *baba*, *babb*, *baaa*, *baaaa*, *baabb*, *baaba*. Can you describe the language consisting of all the words accepted by this automaton?

Note that in the definition of NFA there is no rule that says that for a given state there *must* be an edge for every label. That means that when following a word through a non-deterministic automaton we might find ourselves stuck in a state because there is no edge out of it for the letter we currently see. If that happens then we know that along our current route, the word will not be accepted by the automaton. While this may be confusing at first sight, it is actually quite convenient. It means that pictures of non-deterministic automata can be quite small.

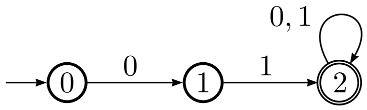
Take the above example of finding an NFA that accepts all the words that start with 0 and end with 1. We can give a much smaller automaton that does the same job.



You should spend a moment convincing yourself that this automaton does indeed accept precisely the words claimed (that is, the same ones as the previous automaton).[3]

This is such a useful convention that it is usually also adopted when *drawing deterministic automata.* Consider the problem of designing a DFA that recognizes those words over the alphabet $\{0, 1\}$ of length at least 2 for which the first letter is 0 and the second letter is 1. By concentrating on what it takes to get a word to an accepting state one might well draw something like this:

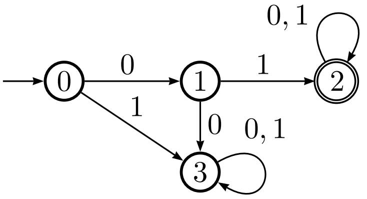
[3] Note that unless we have to describe the automaton in another way, or otherwise have reasons to be able to refer to a particular state, there is no reason for giving the states names in the picture.



This is a perfectly good picture of a deterministic finite automaton. However, not all the states, and not all the transitions, are drawn for this automaton: Above we said that for every state, and every letter from the alphabet, there must be a transition from that state labelled with that letter. Here, however, there is no transition labelled 1 from the state 0, and no transition labelled 0 from the state 1.

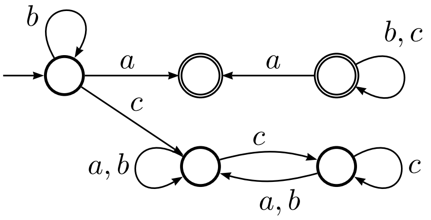
What *does* the automaton do if it sees 1 in state 0, or 0 in state 1? Well, it discards the word as non-acceptable, in a manner of speaking.

We can complete the above picture to show all required states by assuming there's a hidden state that we may think of as a 'dump'. As soon as we have determined that a particular word can't be accepted we send it off in that dump state (which is certainly not an accepting state), and there's no way out of that state. So all the transitions not shown in the picture above go to that hidden state. With the hidden state drawn our automaton looks like this:



This picture is quite a bit more complicated than the previous one, but both describe the same DFA, and so contain precisely *the same information.* I am perfectly happy for you to draw automata either way when it comes to exam questions or assessed coursework.

Exercise 17. Consider the following DFA. Which of its states are dump states, and which are unreachable? Draw the simplest automaton recognizing the same language.



Describe the language recognized by the automaton.

Exercise 18. Go through the automata you have drawn for Exercise 13 and 14. Identify any dump states in them.

3.5 Deterministic *versus* non-deterministic

So far we have found the following differences between deterministic and non-deterministic automata: For the same problem it is usually easier to design a non-deterministic automaton, and the resulting automata are often smaller.

On the other hand, following a word through a deterministic automaton is straightforward, and so deciding whether the word is accepted is easy. For non-deterministic automata we have to find all the possible paths a word might move along, and decide whether any of them leads to an accepting state. Hence finding the language recognized by an NFA is usually harder than to do the same thing for a DFA of a similar size.

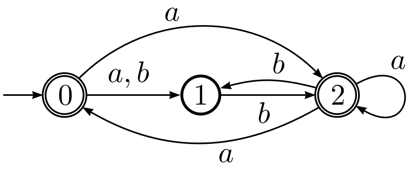
So both have advantages and disadvantages. But how different are they really? Clearly every deterministic automaton can be viewed as a non-deterministic one since it satisfies all the required criteria.

It therefore makes sense to wonder whether there are things we can do with non-deterministic automata that can't be done with deterministic ones. It turns out that this is false.

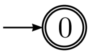
Theorem 3.1. *For every non-deterministic automaton there is a deterministic one that recognizes precisely the same words.*

Algorithm 1, example

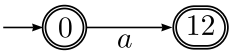
Before looking at the general case of Algorithm 1 we consider an example. Consider the following NFA.



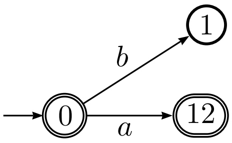
We want to construct a deterministic automaton from this step by step. We start with state 0, which we just copy, so it is both initial and an accepting state in our new automaton.



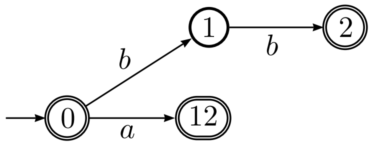
With a, we can go from state 0 to states 1 and 2, so we invent a new state we call 12 (think of it as being a set containing both, state 1 and state 2). Because 2 is an accepting state we make 12 an accepting state too.



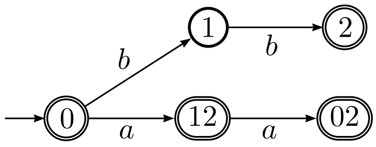
With a letter b we can go from state 0 to state 1, so we need a state 1 (think of it as state $\{1\}$) in the new automaton too.



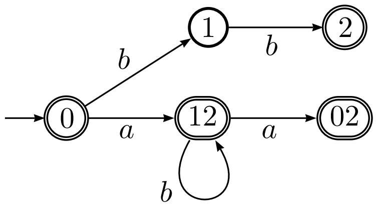
Now we have to consider the states we have just created. In the original automaton from state 1, we can't go anywhere with a, but with b we can go to state 2, so we introduce an accepting state 2 (thought of as $\{2\}$) into our new automaton.



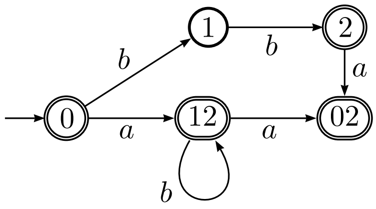
To see where we can go from state 12 we do the following. In the original automaton, with a we can go from 1 nowhere and from 2 to 0 and 2, so we lump these together and say that from 12 we can go to a new state 02 (really $\{0,2\}$) with a. Because 2 is an accepting state we make 02 one too.



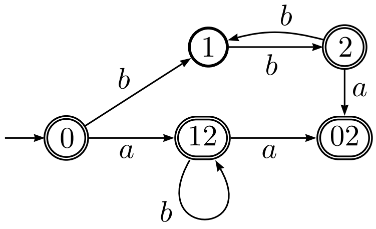
In the original automaton from state 1 with b we can go to state 2, and from state 2 we can go to state 1 with the same letter, so from state 12 we can go back to 12 with b.



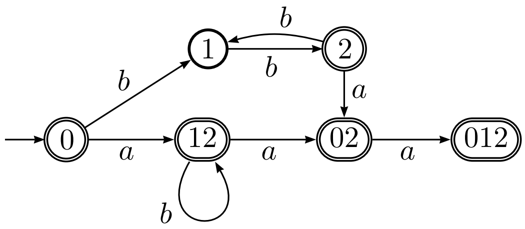
In the original automaton with a we can go from state 2 to states 0 and 2, so we need a transition labelled a from state 2 to state 02 in our new DFA.



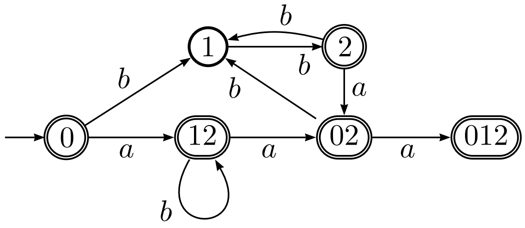
With b from state 2 we can only go back to 1, so we add this transition to the new automaton.



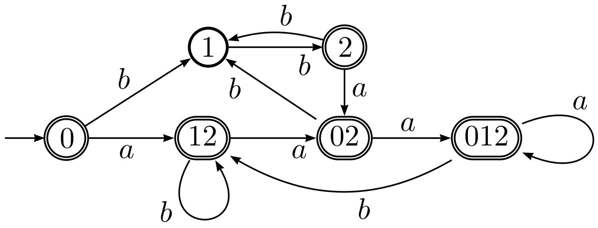
Now for the new state 02. From 0 we can go with a to states 1 and 2, and from state 2 we can get to states 0 and 2, so taking it all together from state 02 we can go to a new accepting state we call 012.



With b from state 0 we can go to state 1 in the old automaton, and from state 2 we can also only go to state 1 with a b, so we need a transition from the new state 02 to state 1 labelled b.



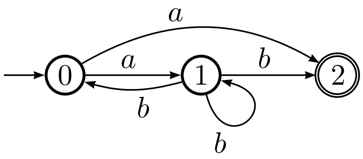
Following the same idea, from 012 with a we can go back to 012, and with b we can go to 12.



So what we have done here? We have created a new automaton from the old one as follows.

- States of the new automaton are *sets of states* from the old one.
- The start state for the new automaton is the set containing just the initial state from the original automaton.
- A state of the new automaton is an accepting state if one of its states in the old automaton is an accepting state.
- There is a transition labelled x from some state in the new automaton to another if there are states of the old automaton in the two sets that have a corresponding transition.

Exercise 19. For the automaton given below carry out the same process. *Hint: If you do find this very hard then read on a bit to see if that helps.*



Algorithm 1, general case

We now give the general case.

Assume that we have an NFA $(Q, q_\bullet, F, \delta)$ over some alphabet Σ. Mathematically, we have constructed a new automaton, a DFA this time, as follows:

- The states are given by $\mathcal{P}Q$, where $\mathcal{P}$ is the powerset operator, so

$$\mathcal{P}Q = \{S \mid S \subseteq Q\}.$$

- The start state is $\{q_\bullet\}$.
- A state S is an accepting state if and only if there is $q \in S$ such that $q \in F$, that is, q is an accepting state in the original automaton. Hence the new set of accepting states is

$$F' = \{S \subseteq Q \mid \exists q \in S \text{ with } q \in F\}.$$

- For a new state $S \in \mathcal{P}Q$ and a letter $x \in \Sigma$ the transition labelled with x leads to the state given by the set of all states q' for which there is a $q \in S$ such that there is a transition labelled x from q to q', that is

$$\{q' \in Q \mid \exists q \in S \text{ with } q \xrightarrow{x} q'\}.$$

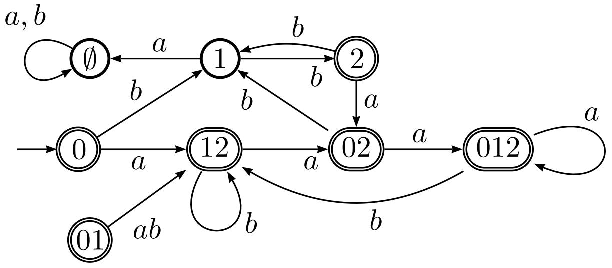
  In other words, to see where the edge labelled x from S should lead to, go through all the states q in S and collect all those states q‘ $\in Q$ which have an edge labelled x from q.

Note that if there is no state in S which is labelled with x then $\delta'(S, x)$ is still defined—it is the empty set in that case. Also note that the state given by the empty set is *always* a ‘dump state’. Because there are no states in $\emptyset$, $\delta(\emptyset, x) = \emptyset$ is true for all $x \in \Sigma$, so it is a state one can never escape from. Also, it cannot be an accepting state because it contains no accepting state as an element.

We call the resulting automaton the **DFA generated by the NFA** $(Q, q_\bullet, F, \delta)$.

For the above example, the new automaton has states $\emptyset$, $\{0\}$, $\{1\}$, $\{2\}$, $\{0, 1\}$, $\{0, 2\}$, $\{1, 2\}$ and $\{0, 1, 2\}$. In the image above we have kept our labels shorter by only listing the elements of the corresponding set without any separators or braces. For larger automata this would make it impossible to distinguish between the state $\{12\}$ and the state $\{1, 2\}$, but in such a small example this is harmless.

However, not all the states we have just listed appear in the picture we drew above. This is so because we used a method that only draws those states which are reachable from the start state. We next draw the full automaton as formally defined above.



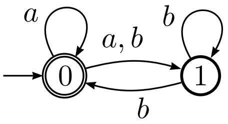
We already know that we may leave out dump states like $\emptyset$ when drawing an automaton.

We have another extra state in this picture, namely $\{0, 1\}$. This is a state we can never get to when starting at the start state $\{0\}$, so no word will ever reach it either. It is therefore irrelevant when it comes to deciding whether or not a word is accepted by this automaton. We call such states **unreachable** and usually don't bother to draw them.

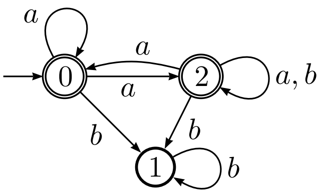
Exercise 20. For the NFA from the previous exercise draw the picture of the full automaton with all states (given by the set of all subsets of $\{0, 1, 2\}$, including the unreachable ones) and all transitions.

Exercise 21. For each of the following NFAs, give a DFA recognizing the same language.

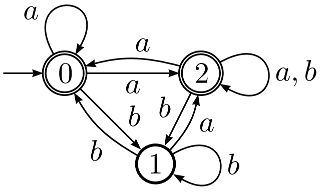
(a)



(b)



(c)



It is possible to avoid drawing unreachable states by doing the following:

- Start with the start state of the new automaton, $\{q_\bullet\}$.
- Pick a state S you have already drawn, and a letter x for which you have not yet drawn a transition out of the state S. Now there must be precisely one state S' to which the transition labelled x leads.
  - If the state S' has already been drawn, draw in the new transition labelled x
  - If the state S' has not been drawn yet, add it to the automaton and draw in the new transition labelled x.

The DFA $(\mathcal{P}Q, \{q_\bullet\}, F', \delta')$ defined above accepts precisely the same words as the original NFA $(\mathcal{P}Q, \{q_\bullet\}, F, \delta)$. This establishes Theorem 3.1, which tells us that we can do precisely the same things with NFAs that we can do with DFAs.

Note that we can use these automata also to encode information that at first sight looks quite different from that of recognizing a language!

Exercise 22. You are asked to design a vending machine that accepts the following coins: 10p, 20p, 50p, 1£. Products handled by the machine cost 1£. In order to keep track of when the machine has received one pound (or more), draw a DFA.

We have three different ways now in which we can describe the same language.

Exercise 23. Consider the following language over the alphabet $\{a, b, c\}$ which consists of all words of odd length. Describe this language in the following ways:

(a) Using a regular expression.

(b) Using a DFA.

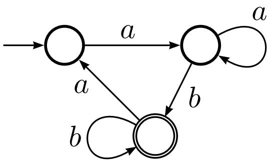
(c) As a set. *Unless you work through the Appendix you will probably find this difficult.*

3.6 From automata to patterns

We have three ways now of talking about collections of strings, or languages. We can describe it set-theoretically, using a regular expression, or using a (deterministic or non-deterministic) finite automaton. The second is particularly good when we have to talk to a computer, and sometimes it's quite easy to come up with the right regular expression. For more complex problems, however, it's often easier (or at least less error-prone) to come up with an automaton. But how do these last two descriptions connect?

If we have found an automaton for the language we want, how do we communicate it to a computer? In this section we describe an algorithm that takes a DFA and turns it into a pattern describing the same language.

Why do we need an algorithm? Sometimes we can just read off the pattern from the automaton, even if they contain cycles.



Clearly any word that will be accepted has to start with a, can then contain arbitrarily many further as, has b, and arbitrarily many further bs. That gets the word into an accepting state. After that, the word may have another a and then it all repeats.

A pattern describing the same language is

$$(aa^*bb^*)(aaa^*bb^*)^*.$$

However, if the automaton is more complicated then reading off a pattern can become very difficult, in particular if there are several accepting states. If the automaton is moreover non-deterministic the complexity of the task worsens further—usually it is therefore a good idea to first convert an NFA to a DFA using Algorithm 1 from page 33.

In order to show that it is possible to construct a regular expression defining the same language for *every* automaton we have to give an algorithm that works for all automata. This algorithm may look overly complex at first sight, but it really does work for every automaton. If you had to apply it a lot you could do the following:

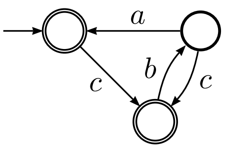
- Define a data structure of finite automata in whatever language you're using. In Java you would be creating a suitable class.

- Implement the algorithm so that it takes as input an object of that class, and produces a pattern accordingly.

Algorithm 2, first example

Because the algorithm required is complicated we explain it first using a simple example. With a bit of experience you will probably be able to read off a pattern from the automaton, but you won't be able to do this with more complicated automata (nor will you be able to program a computer to do the same).

Consider the following DFA.



Before we can apply the algorithm we have to name the states, and the algorithm assumes that the states are, in fact *numbered.* The **order** in which the states are numbered can make a big difference in **how complex** the algorithm is to carry out! Turning an automaton into a regular expression is difficult to do in general since *every loop* in the automaton has to be taken into account. You therefore want to apply the following rules:

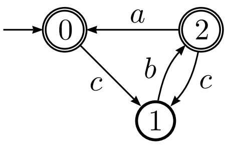
- Count the number of states, say there are n. We number them from 0 to $n-1$.
- Assume we have reached number $n-i$.
  - Find the state not yet numbered that is involved in the most loops not yet eliminated. Label that state $n-i-1$.
  - Remove all those loops from consideration that involve the newly named state.

Or, if you are looking for something simpler:

- Number the states in *reverse* order of complexity,

that is, the state that has the most paths going through it gets the highest number, the next highest number goes to the next 'complicated' state, and so on. If states look roughly similar then it doesn't matter.

In the example above, the two non-initial states are involved in more paths, but they're fairly similar, so it doesn't matter which of them gets the higher number. We use the following numbering.



There are two accepting states. Hence a word that is accepted will either start in state 0 and end in state 0 or start in state 0 and end in state 2. That means the language $\mathcal{L}$ accepted by this automaton is the union of two languages which we write as follows:

$$\mathcal{L} = \mathcal{L}_{0\to 0} \cup \mathcal{L}_{0\to 2}$$

The indices tell us in which state we start and in which state we finish. This is already a useful observation since we can now concentrate on calculating one language at a time.

To calculate $\mathcal{L}_{0\to 0}$ we note that we can move from state 0 to state 0 in a number of ways. It is so complicated because there are loops in the automaton. What we do is to break up these ways by controlling the use of the state with the highest number, state 2, as follows:

To get from state 0 to state 0 we can

- either not use state 2 at all (that is, go only via states 0 and 1) or
- go to state 2, return to it as many times as we like, and then go from state 2 to state 0 at the end.

At first sight this does not seem to have simplified matters at all. But we have now gained control over how we use the state 2 because we can use this observation to obtain the following equality.

$$\mathcal{L}_{0\to 0} = \mathcal{L}^{\leq 2}_{0\to 0} = \mathcal{L}^{\leq 1}_{0\to 0} \cup \mathcal{L}^{\leq 1}_{0\to 2} \cdot (\mathcal{L}^{\leq 1}_{2\to 2})^* \cdot \mathcal{L}^{\leq 1}_{2\to 0}$$

We have introduced new notation here, using $\mathcal{L}^{\leq 1}_{i\to j}$ to mean that we go from state i to state j while using only states 0 and 1 in between, that is, states with a number ≤ 1. While our expression has grown the individual languages are now easier to calculate.

- $\mathcal{L}^{\leq 1}_{0\to 0}$. We cannot go from state 0 to state 0 while only using states 0 and 1 in between, other than just staying in state 0, which corresponds to following the empty word ϵ through the automaton. Hence this language is equal to $\{\epsilon\}$.
- $\mathcal{L}^{\leq 1}_{0\to 2}$. To go from state 0 to state 2 without using state 2 in between we must see c followed by b. Hence this language is equal to $\{cb\}$.
- $\mathcal{L}^{\leq 1}_{2\to 2}$. To go from state 2 to state 2 without using state 2 in between there are two possibilities:
  - we can go from state 2 to state 1 and back, seeing cb along the way or
  - we can go from state 2 to state 0 to state 1 to state 2 and see acb along the way. (It would be a good idea now to convince yourself that there is no other possibility.)

  Hence this language is equal to $\{cb\} \cup \{acb\} = \{cb, acb\}$.
- $\mathcal{L}^{\leq 1}_{2\to 0}$. The only way of getting from state 2 to state 0 using only states 0 and 1 in between is the direct route, which means we must see a. Hence this language is equal to $\{a\}$.

Putting all these together we get the following.

$$\begin{aligned}\mathcal{L}_{0\to 0} = \mathcal{L}^{\leq 2}_{0\to 0} &= \mathcal{L}^{\leq 1}_{0\to 0} \cup \mathcal{L}^{\leq 1}_{0\to 2} \cdot (\mathcal{L}^{\leq 1}_{2\to 2})^* \cdot \mathcal{L}^{\leq 1}_{2\to 0} \\ &= \{\epsilon\} \cup \{cb\} \cdot \{cb, acb\}^* \cdot \{a\}\end{aligned}$$

We can now read a pattern[4] off from this expression, namely

$$\boldsymbol{\epsilon}|cb(cb|acb)^*a,$$

and we have

$$\mathcal{L}_{0\to 0} = \mathcal{L}(\boldsymbol{\epsilon}|cb(cb|acb)^*a).$$

Note that this is not the simplest pattern we could possibly have used—$(c(bc)^*ba)^*$ would also have worked. The advantage of the algorithm is that it *always* gives you a pattern in the end, no matter how complicated the automaton.

In this case the languages we were required to calculate were not so difficult to work out. If we had started with a more complicated automaton we would have had to employ the same trick again, namely taking another state and controlling how that is used. The second example shows how this works.

We do yet have to calculate the second language, $\mathcal{L}_{0\to 2}$. Again we do so by controlling when we use state 2, but this time it is slightly easier:

To get from state 0 to state 2 we can

- go to state 2 and return to it as many times as we like.[5]

This leads us to the following equality.

$$\mathcal{L}_{0\to 2} = \mathcal{L}^{\leq 2}_{0\to 2} = \mathcal{L}^{\leq 1}_{0\to 2} \cdot (\mathcal{L}^{\leq 1}_{2\to 2})^*.$$

Moreover, we already have worked out the languages required here, so we can continue directly to write the following.

$$\begin{array}{rcl}\mathcal{L}_{0\to 2} & = & \mathcal{L}^{\leq 1}_{0\to 2} \cdot (\mathcal{L}^{\leq 1}_{2\to 2})^* \\ & = & \{cb\} \cdot \{cb, acb\}^*\end{array}$$

From this we can read off a regular expression, namely $cb(cb|acb)^*$, and we get

$$\mathcal{L}_{0\to 2} = \mathcal{L}(cb(cb|acb)^*).$$

Altogether that means

$$\begin{array}{rcl}\mathcal{L} & = & \mathcal{L}_{0\to 0} \cup \mathcal{L}_{0\to 2} \\ & = & \mathcal{L}(\boldsymbol{\epsilon}|cb(cb|acb)^*a) \cup \mathcal{L}(cb(cb|acb)^*) \\ & = & \mathcal{L}(\boldsymbol{\epsilon}|cb(cb|acb)^*a|cb(cb|acb)^*)\end{array}$$

If we like we can simplify that as follows.

$$\mathcal{L}(\boldsymbol{\epsilon}|cb(cb|acb)^*a|cb(cb|acb)^*) \quad = \quad \mathcal{L}(\boldsymbol{\epsilon}|cb(cb|acb)^*(a|\boldsymbol{\epsilon}))$$

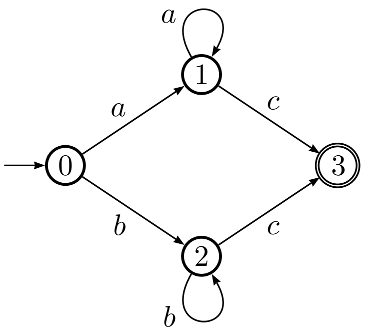
Alternatively, with a bit of experience one might read off this regular expression directly from the automaton:

$$(c(bc)^*ba)^*(\boldsymbol{\epsilon}|c(bc)^*b)$$

[4] If you find this difficult go back and do Exercises 1, 2, and 8 and have another look at page 18.

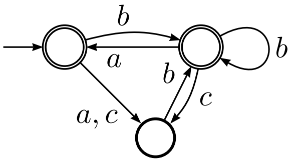
[5] It would be a good idea to work out why this is different from the case of language $\mathcal{L}_{0\to 0}$.

Exercise 24. Carry out the algorithm just described for the following automaton. Use the notation with the various languages $\mathcal{L}_{j \to i}^{\leq k}$ for the first two or three lines at least, even if you can read off a pattern directly. If you want to compare your solution to the model answers you should adopt the numbering given in the picture.

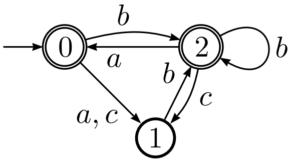


Algorithm 2, second example

Let's look at a more complicated example.[6]



Again we have to number the states. The most 'complicated' state is the one on the top right, and once that is removed there is not much difference, so we use the following numbering.



What is confusing about this automaton is not its size in terms of the number of states, but the way the transitions criss-cross it.

In order to find all the words accepted by the automaton we have to identify all the words that

- when starting in state 0 end up in state 0. We call the resulting language $\mathcal{L}_{0 \to 0}$. We also need all the words that
- when starting in state 0 end up in state 2. We call the resulting language $\mathcal{L}_{0 \to 2}$.

In other words the language $\mathcal{L}$ recognized by the automaton can be calculated as

$$\mathcal{L} = \mathcal{L}_{0 \to 0} \cup \mathcal{L}_{0 \to 2}.$$

This now allows us to calculate these two languages separately, which already makes the task a little bit easier. Note that what we have done is to

[6] Note that the automaton from the first example is a 'sub-automaton' of this one—all we have done here is to add some further transitions.

split up the recognized language into all the languages of the form $\mathcal{L}_{q_\bullet \to q}$, where $q_\bullet$ is the start state and q ranges over all the accepting states (that is all the elements of F).

In general we would therefore write for the language $\mathcal{L}$ recognized by some automaton $(Q, q_\bullet, F, \delta)$

$$\mathcal{L} = \bigcup_{q \in F} \mathcal{L}_{q_\bullet \to q}.$$

But finding all the different paths that lead from 0 to 0, or from 0 to 2 is still pretty tough. The way we simplify that is by taking the state with the highest index, namely 2, out of consideration as follows.

Every path from the state 0 to the state 0 can do one of the following:

- It either doesn't use the state 2 at all or
- it goes from the state 0 to the state 2, then goes back to the state 2 as often as it likes, and ultimately goes to the state 0.

At first sight this doesn't look like a very useful observation. But what we have done now is to break up any path that starts at the state 0 and finishes at the state 0 into a succession of paths that *only use the state* 2 *at controlled points.*

We use the same notation as before: All words that follow a path that goes from state 0 to state 0 while only using states 0 and 1 (but not state 2) in between make up the language $\mathcal{L}^{\leq 1}_{0\to 0}$. This works similarly for other start or end states. Reformulating our last observation means then that every word that follows a path from state 0 to state 0 satisfies one of the following:

- It either is an element of $\mathcal{L}^{\leq 1}_{0\to 0}$ or
- it is an element of
$$\mathcal{L}^{\leq 1}_{0\to 2} \cdot (\mathcal{L}^{\leq 1}_{2\to 2})^* \cdot \mathcal{L}^{\leq 1}_{2\to 0}.$$

That means that we have the equality

$$\mathcal{L}_{0\to 0} = \mathcal{L}^{\leq 2}_{0\to 0} = \mathcal{L}^{\leq 1}_{0\to 0} \cup \mathcal{L}^{\leq 1}_{0\to 2} \cdot (\mathcal{L}^{\leq 1}_{2\to 2})^* \cdot \mathcal{L}^{\leq 1}_{2\to 0}.$$

While the equality may appear to make things more confusing at first sight, we now have languages which we can more easily determine on the right hand side of the equality.

We now have the choice between trying to determine the languages on the right directly, or applying the same idea again.

- $\mathcal{L}^{\leq 1}_{0\to 0}$. How do we get from state 0 to state 0 only using states 0 and 1? The simple answer is that we can't move there, but we are already there so $\mathcal{L}^{\leq 1}_{0\to 0} = \{\epsilon\}$.
- $\mathcal{L}^{\leq 1}_{0\to 2}$. Going from state 0 to state 2 using only states 0 and 1 can be done in two ways, either directly using the letter b or *via* state 1 using ab or cb. Hence $\mathcal{L}^{\leq 1}_{0\to 2} = \{b, ab, cb\}$.
- $\mathcal{L}^{\leq 1}_{2\to 2}$. This is more complicated. Instead of trying to work this out directly we apply our rule again: When going from state 2 to state 2 using only states 0 and 1 we can either go directly from state 2 to

state 2 or we can go from state 2 to state 1, return to state 1 as often as we like using only state 0, and then go from state 1 to state 2. In other words we have.

$$\mathcal{L}^{\leq 1}_{2\to 2} = \mathcal{L}^{\leq 0}_{2\to 2} \cup \mathcal{L}^{\leq 0}_{2\to 1} \cdot (\mathcal{L}^{\leq 0}_{1\to 1})^* \cdot \mathcal{L}^{\leq 0}_{1\to 2}.$$

We now read off $\mathcal{L}^{\leq 0}_{2\to 2} = \{b, ab\}$, $\mathcal{L}^{\leq 0}_{2\to 1} = \{aa, ac, c\}$, $\mathcal{L}^{\leq 0}_{1\to 1} = \{\epsilon\}$ and $\mathcal{L}^{\leq 0}_{1\to 2} = \{b\}$. That gives us

$$\begin{aligned}\mathcal{L}^{\leq 1}_{2\to 2} &= \{b, ab\} \cup \{aa, ac, c\} \cdot \{\epsilon\}^* \cdot \{b\}\\ &= \{b, ab\} \cup \{aa, ac, c\} \cdot \{\epsilon\} \cdot \{b\}\\ &= \{b, ab, aab, acb, cb\}.\end{aligned}$$

One step in this calculation merits further explanation, namely $\emptyset^*$, which is calculated to be $\{\epsilon\}$ on page 13.

- $\mathcal{L}^{\leq 1}_{2\to 0}$. This one we can read off directly, it is $\{a\}$.

Altogether we get

$$\begin{aligned}\mathcal{L}_{0\to 0} &= \{\epsilon\} \cup \{b, ab, cb\} \cdot \{b, ab, aab, acb, cb\}^* \cdot \{a\}\\ &= \mathcal{L}(\boldsymbol{\epsilon}) \cup \mathcal{L}(b|ab|cb) \cdot (\mathcal{L}(b|ab|aab|acb|cb))^* \cdot \mathcal{L}(a)\\ &= \mathcal{L}(\boldsymbol{\epsilon}|(b|ab|cb)(b|ab|aab|acb|cb)^*a).\end{aligned}$$

See the top of page 18 for an explanation of the last step.

This leaves the calculation of $\mathcal{L}_{0\to 2}$. We once again apply the earlier trick and observe this time that a path from state 0 to state 2 will have to reach state 2 (for the first time) using only states 0 and 1 along the way, and then it can return to state 2 as often as it likes, giving

$$\mathcal{L}_{0\to 2} = \mathcal{L}^{\leq 2}_{0\to 2} = \mathcal{L}^{\leq 1}_{0\to 2} \cdot (\mathcal{L}^{\leq 1}_{2\to 2})^*.$$

Now $\mathcal{L}^{\leq 1}_{0\to 2}$ we already calculated above, it is equal to $\{b, ab, cb\}$. We also know already that $\mathcal{L}^{\leq 1}_{2\to 2} = \{b, ab, aab, acb, cb\}$. Hence

$$\mathcal{L}_{0\to 2} = \{b, ab, cb\} \cdot \{b, ab, aab, acb, cb\}^* = \mathcal{L}((b|ab|cb)(b|ab|aab|acb|cb)^*).$$

Hence the language recognized by the automaton is

$$\begin{aligned}&\mathcal{L}_{0\to 0} \cup \mathcal{L}_{0\to 2}\\ &\quad= \mathcal{L}(\boldsymbol{\epsilon}|(b|ab|cb)(b|ab|aab|acb|cb)^*a) \cup \mathcal{L}((b|ab|cb)(b|ab|aab|acb|cb)^*)\\ &\quad= \mathcal{L}(\boldsymbol{\epsilon}|((b|ab|cb)(b|ab|aab|acb|cb)^*a)|((b|ab|cb)(b|ab|aab|acb|cb)^*))\\ &\quad= \mathcal{L}(\boldsymbol{\epsilon}(b|ab|cb)(b|ab|aab|acb|cb)^*(\boldsymbol{\epsilon}|a)),\end{aligned}$$

and a regular expression giving the same language is

$$\boldsymbol{\epsilon}|(b|ab|cb)(b|ab|aab|acb|cb)^*(\boldsymbol{\epsilon}|a).$$

Algorithm 2, general case

It is time to generalize what we have done into an algorithm, known as Algorithm 2. It is useful to assume that the states in the automaton are given by $\{0, 1, \ldots, n\}$, and that 0 is the start state. If we get an automaton that does not conform with this we can always relabel the states accordingly. Then the language $\mathcal{L}$ of all the words accepted by the automaton is equal to

$$\bigcup_{i\in F} \mathcal{L}_{0\to i} = \bigcup_{i\in F} \mathcal{L}^{\leq n}_{0\to i},$$

where $\mathcal{L}_{0\to i}$ is the language of all words that, when starting in state 0 end in state i. Since a word is accepted if and only if it ends in an accepting state the above equality is precisely what we need.

We can now think of some $\mathcal{L}_{0\to i}$ as equal to $\mathcal{L}^{\leq n}_{0\to i}$: the language of all words that, when starting in state 0, end up state i is clearly the language of all words that do so when using any of the states in $\{0, 1, \ldots, n\}$. In general, $\mathcal{L}^{\leq k}_{j\to i}$ is the language of all those words that, when starting in state j end in state i, use only states with a number less than or equal to k in between. It is the languages of the form $\mathcal{L}^{\leq k}_{j\to i}$ for which we can find expressions that reduce k by 1: Any path that goes from state j to state i using only states with numbers at most k will

- either go from state j to state i only using states with number at most $k-1$ in between (that is, not use state k at all)
- or go from state j to state k (using only states with number at most $k-1$ in between), return to state k an arbitrary number of times, and then go from state k to state i using only states with number at most $k-1$ in between.

Hence we have

$$\mathcal{L}^{\leq k}_{j\to i} = \mathcal{L}^{\leq k-1}_{j\to i} \cup \mathcal{L}^{\leq k-1}_{j\to k} \cdot (\mathcal{L}^{\leq k-1}_{k\to k})^* \cdot \mathcal{L}^{\leq k-1}_{k\to i}.$$

We note that if $j = k$ or $i = k$ then in the above expression only two different languages appear, namely

$$\text{if } j = k \qquad \text{then} \qquad \mathcal{L}^{\leq k-1}_{j\to i} = \mathcal{L}^{\leq k-1}_{k\to i} \qquad \text{and} \qquad \mathcal{L}^{\leq k-1}_{j\to k} = \mathcal{L}^{\leq k-1}_{k\to k}$$

and

$$\text{if } i = k \qquad \text{then} \qquad \mathcal{L}^{\leq k-1}_{j\to i} = \mathcal{L}^{\leq k-1}_{j\to k} \qquad \text{and} \qquad \mathcal{L}^{\leq k-1}_{k\to i} = \mathcal{L}^{\leq k-1}_{k\to k}.$$

Thus we get slightly simpler expressions (compare $\mathcal{L}_{0\to 2}$ in the above example):

$$\begin{array}{rcl} \mathcal{L}^{\leq j}_{j\to i} & = & (\mathcal{L}^{\leq j-1}_{j\to j})^* \mathcal{L}^{\leq j-1}_{j\to i} \\ \mathcal{L}^{\leq i}_{j\to i} & = & \mathcal{L}^{\leq i-1}_{j\to i} (\mathcal{L}^{\leq i-1}_{i\to i})^* \end{array}$$

Because the number of states involved in the paths in questions goes down the languages on the right hand side of the equality are easier to calculate. If we have to we can keep applying the above equality until we have reached languages where the only state allowed on the way is a state with number below 0—that is, no state at all. Of course the overall expression we have will become longer and longer, so it pays to read off languages from the automaton as soon as that is possible.

Once we have reached paths which are not allowed to use any other states we have the following.

For $j = i$:

$$\mathcal{L}^{\leq -1}_{i \to i} = \begin{cases} \{\epsilon\} & \text{if there is no transition from } i \text{ to } i \\ \{\epsilon, x_1, x_2, \dots x_l\} & \text{if the } x_i \text{ are all the labels on the} \\ & \text{transition from } i \text{ to itself.} \end{cases}$$

For $j \neq i$:

$$\mathcal{L}^{\leq -1}_{j \to i} = \begin{cases} \emptyset & \text{if there is no transition from } j \text{ to } i \\ \{x_1, x_2, \dots x_l\} & \text{if these are all the labels on the} \\ & \text{transition from } j \text{ to } i. \end{cases}$$

There are a number of equalities that allow us to simplify the expression we obtain in this way.

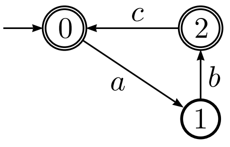
- $\emptyset \cup \mathcal{L} = \mathcal{L} = \mathcal{L} \cup \emptyset$;
- $\emptyset^* = \{\epsilon\} = \{\epsilon\}^*$;
- $\emptyset \cdot \mathcal{L} = \emptyset = \mathcal{L} \cdot \emptyset$;
- $\{\epsilon\} \cdot \mathcal{L} = \mathcal{L} = \mathcal{L} \cdot \{\epsilon\}$
- $(\{\epsilon\} \cup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

Once we have an expression consisting entirely of the simple languages of the form $\mathcal{L}^{\leq -1}_{j \to i}$ we can insert regular expressions generating these, and use the rules from page 18 to get one regular expression for the overall language.

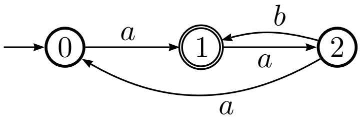
The advantage of having a general algorithm that solves the problem is that one can now write a program implementing it that will work for all automata, even large ones.

Exercise 25. Using Algorithm 2 find a pattern for the following automata. Can you read off a pattern from each automaton? Does it match the one constructed?

(a)

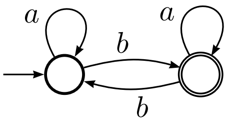


(b)

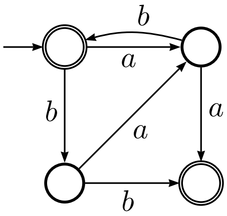


Exercise 26. Give regular expressions defining the languages recognized by the following automata using Algorithm 2. *Hint: Recall that the way you number the states has an impact on how many steps of the algorithm you will have to apply!*

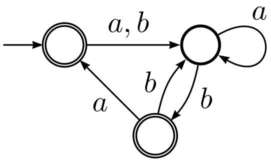
(a)



(b)



(c)



By giving the algorithm we have established the following result.

Theorem 3.2. *For every DFA there is a regular expression such that the language recognized by the DFA is the language defined by the regular expression.*

We get the following consequence thanks to Theorem 3.1.

Theorem 3.3. *For every DFA or NFA it is the case that the language it recognizes is regular.*

3.7 From patterns to automata

We have a way of going from an automaton to a pattern that we can communicate to a computer, so a natural question is whether one can also go in the opposite direction. This may sound like a theoretical concern at first sight, but it is actually quite useful to be able to derive an automaton from a pattern. That way, if one does come across a pattern that one doesn't entirely understand one can turn it into an automaton. Also, changing existing patterns so that they apply to slightly different tasks can often be easier done by first translating them to an automaton.

For some patterns we can do this quite easily

Exercise 27. Design DFAs over the alphabet $\{a, b, c\}$ that recognize the languages defined by the following patterns.

(a) $(a|b)cc$.

(b) $cc(a|b)$.

(c) $aa|bb|cc$.

(d) $c(a|b)^*c$.

Now assume that instead we want to recognize all words that contain a substring matching those patterns. How do you have to change your automata to achieve that?

For slightly more complicated patterns we can still do this without too many problems, provided we are allowed to use NFAs.

Exercise 28. Design NFAs over the language $\{0,1\}$ that recognize the languages defined by the following patterns.

(a) $(00)^*|(01)^*$

(b) $(010)^*|0(11)^*$

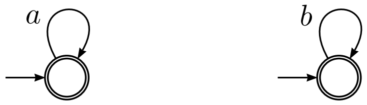
Now turn your NFAs into DFAs.

However, in general it can be quite difficult to read off an automaton from a pattern.[7] We therefore introduce an algorithm that works for all regular expression. This algorithm is *recursive*, and it builds on Definition 1, making use of the recursive structure of patterns. It is very easy to build automata for the base cases. However, to build automata for the constructors of patterns alternative, concatenation, and star, we need to be a bit cleverer.

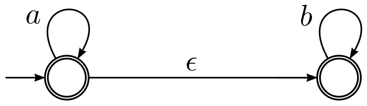
Algorithm 3, example

Assume that we want to build an automaton for the regular expression a^*b^* based on already having automata for the patterns a^* and b^*.

The latter are quite easily constructed:



Now we would somehow like to stick these automata together, so that the first part accepts the a^* part of our word, and the second part the b^* part. For that we'd like to have a way of going from the left state above to the right state above in a way that doesn't consume any of the letters of our word, something like this:



That then is precisely what we do: We generalize our notion of automaton to include transitions labelled not by a letter from our alphabet, but by the empty word ϵ.

Definition 10. *Let Σ be an alphabet not containing ϵ. An **NFA with ϵ-transitions over** Σ is an NFA over the alphabet Σ that may have transitions labelled with ϵ. Hence the transition relation δ relates pairs of the form (q,x), where q is a state and x is either an element of Σ or equal to ϵ, to states.*

We now have to worry about what it means for an NFA with ϵ-transitions to accept a word. Whenever there is a transition labelled with ϵ in the automaton we are allowed to follow it without matching the next letter in our word.

[7] You can always write down some reasonably complex patterns and give it a go.

Definition 11. *A word $s = x_1 \cdots x_n$ over Σ* ***is accepted by the NFA with ϵ-transitions*** *$(Q, q_\bullet, F, \delta)$* ***over*** *$\Sigma$ if there are states*

$$q_0 = q_\bullet, q_1, \ldots, q_l$$

and, for all $1 \leq i \leq n$, transitions

$$q_{m_{i-1}} \xrightarrow{\epsilon} q_{m_{i-1}+1} \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} q_{m_i-1} \xrightarrow{x_i} q_{m_i}$$

as well as transitions

$$q_{m_n} \xrightarrow{\epsilon} q^{1}_{m_n+1} \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} q_l,$$

such that $m_0 = 0$ and such that q_l is an accepting state. Here in each case the number of ϵ-transitions may be 0. The ***language recognized by an NFA with ϵ-transitions*** *is the set of all words it accepts.*

While we do need NFAs with ϵ-transitions along the way to constructing an automaton from a pattern we do not want to keep the ϵ-transitions around since they make the automata in question much more confusing. We introduce an algorithm here that removes the ϵ-transitions from such an automaton. For that let Σ be an arbitrary alphabet not containing ϵ.

Before we turn to describing the **general case of Algorithm 3** we investigate the algorithm that removes ϵ-transitions.

Algorithm 4, general case

Algorithm 4 is a case where it makes sense to give the general case and then looking at an example.

Let $(Q, q_\bullet, F, \delta)$ be an NFA with ϵ-transitions over some alphabet Σ. We define an NFA $(Q, q_\bullet, F', \delta')$ without ϵ-transitions over the same alphabet as follows:

- Let

  $$F' = F \cup \{q \in Q \mid \exists q = q_0, \ldots, q_n \in Q \text{ with } q_i \xrightarrow{\epsilon} q_{i+1} \text{ for all } 0 \leq i \leq n-1 \text{ and } q_n \in F\}.$$

  In other words we add to F those states for which we can get to an accepting state by following only ϵ moves.

- For $q, q' \in Q$ and $x \in \Sigma$, the transition relation δ' relates (q, x) to q' if and only if there are states $q = q_1, q_2 \ldots q_n = q'$ such that

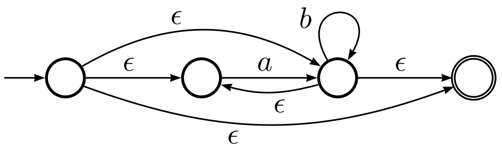
  $$q = q_1 \xrightarrow{\epsilon} q_2 \cdots q_{n-2} \xrightarrow{\epsilon} q_{n-1} \xrightarrow{x} q_n = q'.$$

  In other words for $x \in \Sigma$ and $q, q' \in Q$ there is a transition labelled x from q to q' in δ' if it is possible to get from q to q' in δ by following an arbitrary number of ϵ-transitions followed by one transition labelled x.

The resulting automaton may then contain unreachable states that can be safely removed. It recognizes precisely the same language as the automaton we started with.

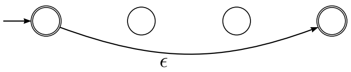
Algorithm 4, example

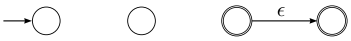
Here is an example. Consider the following automaton with ϵ-transitions.



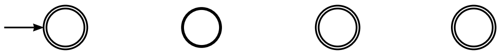
We copy the states as they are, and create some new accepting states as follows:

Pick a non-accepting state. If from there we can reach an accepting state (in the original automaton) *using only ϵ-transitions*, we make this state an accepting state.

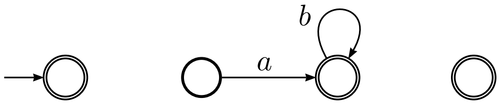




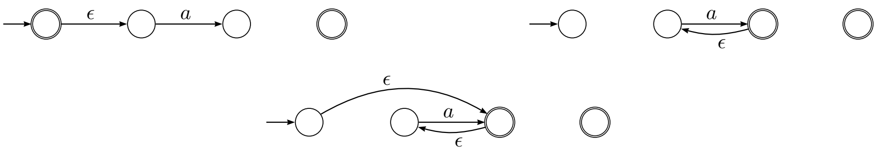
For the example, this means the initial state and the third state from the left are now accepting states.



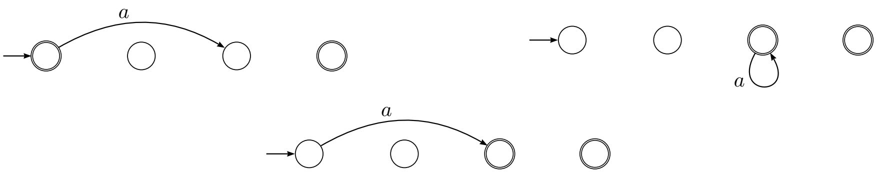
We then copy all the transitions labelled with a letter other than ϵ from the original automaton.



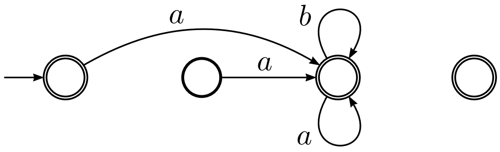
We now look at each of these transitions in turn. Take the transition labelled a. We add a transition labelled a from state i to state j if it is possible to get from state i to state j using a number of ϵ-transitions followed by the transition labelled a.



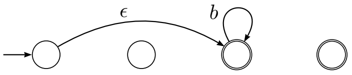
These give rise to the following transitions for our automaton:



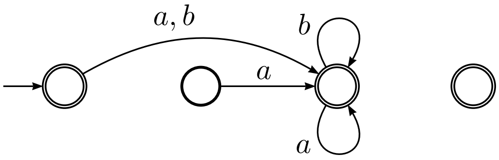
This gives us two new transitions included below.



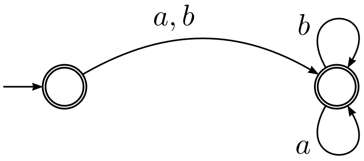
We now do the same for the transition labelled b.



This gives us one new transition, but we already have a transition from the initial state to the third state from the left, so we can just add the new label b to that.[8]

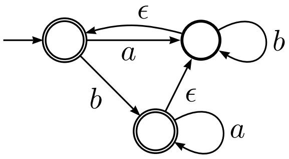


Now we may safely remove the unreachable states and any transitions involving them.



We now have an automaton without ϵ-transitions that accepts precisely the same words as the origin.

Exercise 29. Turn the following automaton into one that does not have ϵ-transitions.



There is further opportunity for practising this algorithm below.

Theorem 3.4. *For every NFA with ϵ-transitions there is an NFA over the same alphabet that recognizes the same language.*

Algorithm 3, general case

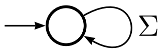
We are now ready to look at the process by which we recursively turn a pattern over some alphabet Σ into an NFA with ϵ-transitions over the same alphabet, known as Algorithm 3.[9]

[8] We could also make do with just one loop transition being drawn—I only left in two to make the process clearer.

[9] Go back to page 17 if you can't see how the following works.

The pattern $\emptyset$

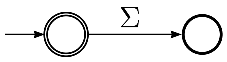
An automaton recognizing the language defined by this pattern is given below.



An automaton that accepts no word at all.[10]

The pattern ϵ

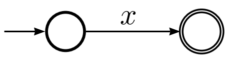
An automaton recognizing the language defined by this pattern is given below.



An automaton that accepts precisely the word ϵ.[11]

The pattern x for $x \in \Sigma$

An automaton recognizing the language defined by this pattern is given below.

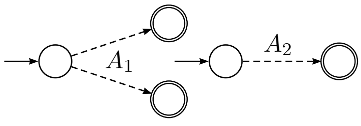


An automaton that accepts precisely the word x.

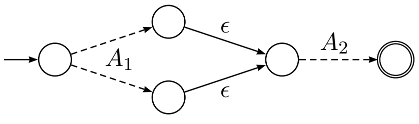
For the next two cases assume that we have patterns p_1 and p_2 as well as automata A_1 and A_2 such that A_i recognizes the language defined by p_i.

Concatenation

An automaton recognizing the language defined by the pattern $p_1 p_2$ is given below. Here we turn every accepting state of A_1 into a non-accepting state and draw an ϵ-transition from it to the start state of A_2. The only accepting states in the new automaton are those from A_2.



combine to

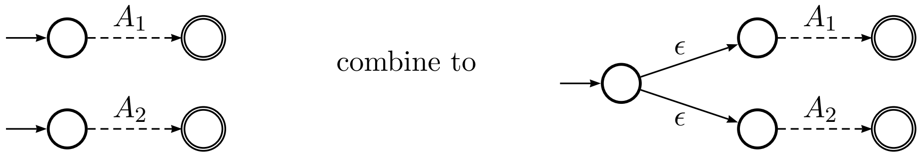


[10] We could leave out the transition entirely.

[11] Again we could leave out the transition and the right hand state, which is a dump state.

Alternative

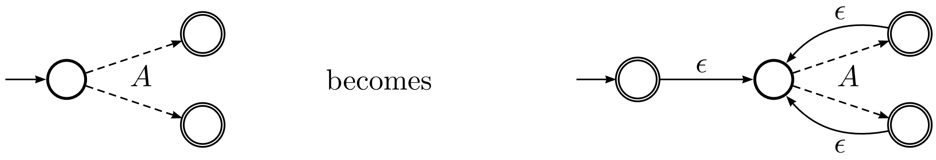
An automaton accepting the language defined by the pattern $p_1|p_2$ is given below. We add a new start state and connect it with ϵ-transitions to the start states of A_1 and A_2 (so these are no longer start states).



Kleene Star

We assume that we have a pattern p and an automaton A that recognizes the language defined by p.

An automaton accepting the language defined by the pattern p^* is given below. Given an automaton A we introduce a new start state. This state is accepting, and it has an ϵ-transition to the old start state. Further we introduce an ϵ-transition from each accepting state to the old start state.



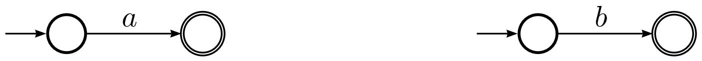
Exercise 30. Refer to your solutions to Exercise 27 to do the following. Design NFAs with ϵ-transitions for the following:

(a) All words over $\{a, b, c\}$ that match the pattern $(aa|bb|cc)(a|b)cc$.

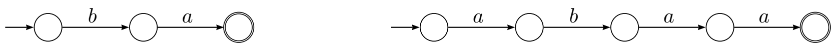
(b) All words over $\{a, b, c\}$ that match the pattern $((a|b)cc)|(cc(a|b))$.

(c) All words over $\{a, b, c\}$ that match the pattern $(aa|bb|cc)^*$.

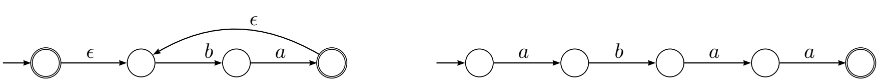
We illustrate the process with an example. Consider $(a(ba)^*|abaa)^*$. We construct an automaton that accepts the language given by that pattern. For the patterns a and b we have the automata



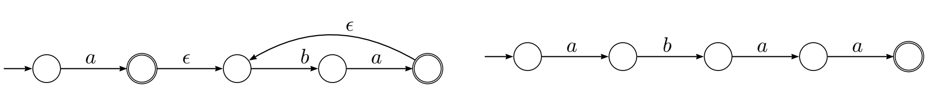
Hence for ba and $abaa$, respectively, we have



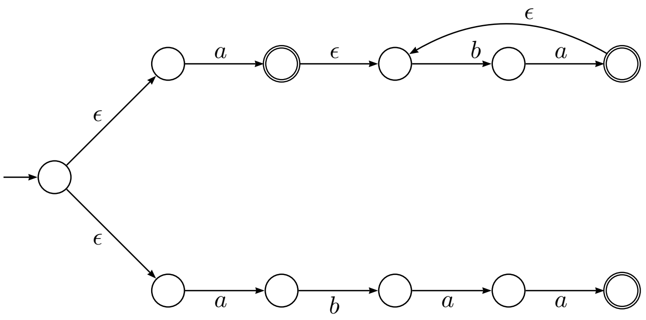
For the left hand automaton we apply the Kleene star, so on the left there's an automaton for $(ba)^*$ now.



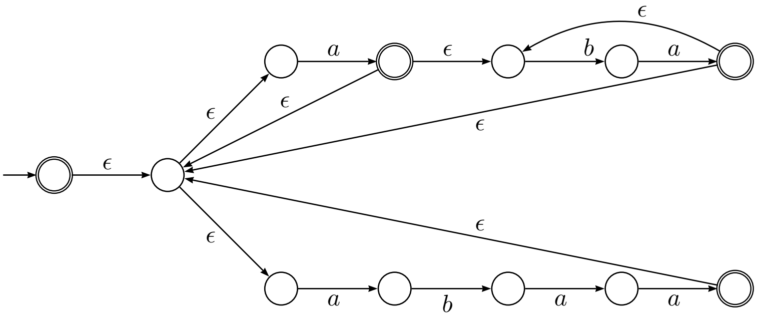
In another step we turn the left hand automaton into one for $a(ba)^*$.



We now combine the two automata using the rule for alternative to obtain the following automaton for $(a(ba)^*|abaa)$:



It remains to apply the Kleene star to the whole automaton in order to obtain one for $(a(ba)^*|abaa)^*$.



Note that this automaton has more ϵ-transitions than required, and in practice it is fine to leave out ones that are clearly redundant. Note that we have constructed an NFA with ϵ-transitions, so we have to apply Algorithm 4 to remove the ϵ-transitions, and then Algorithm 1 to obtain a DFA.

In practice strictly following Algorithm 3 from page 3.7 introduces rather more ϵ-transitions than strictly necessary. In particular when concatenating very simple automata, the ϵ-transition connecting them can often be safely left out. I am happy for you to do so in assessed work.

Exercise 31. Carry out the following tasks.

(a) Use Algorithm 3 described on page 48 to construct an NFA with ϵ-transitions that recognizes the language defined by the pattern $(a|b)^*a(a|b)$.

(b) Take your result from (a) and remove the ϵ-transitions by applying Algorithm 4 on page 46. Now remove all unreachable states.

(c) Take your result from (b) and turn it into a DFA by using Algorithm 1 on page 33. How many states would your DFA have if you didn't remove unreachable states first? How many states would your DFA have if you didn't only draw the reachable ones?

Exercise 32. Construct DFAs that recognize the languages defined by the following patterns. You may either just write down automata, or use Algorithms 3, 4 and 1 to do this.

(a) $(ab|a)^*$

(b) $(01)^*|001^*$

(c) $((00)^*11|01)^*$

(d) $((ab)^*b|ab^*)^*$

We have given an idea of how to prove the following.

Theorem 3.5. *For every regular language we can find the following recognizing it:*

- *deterministic finite automaton;*
- *non-deterministic finite automaton;*
- *non-deterministic finite automaton with ϵ-transitions.*

Altogether, we have shown in various parts of this section the following result.

Theorem 3.6. *The following have precisely the same power when it comes to describing languages:*

- *regular expressions,*
- *deterministic finite automata,*
- *non-deterministic finite automata.*

Hence for every regular language we can find a description using any one of the above, and given one description we can turn it into any of the others.

3.8 Properties of regular languages

In Section 2 there are examples of how we can build new regular languages from existing ones. At first sight these may seem like theoretical results without much practical use. However, what they allow us to do is to build up quite complicated languages from simpler ones. This also means that we can build the corresponding regular expressions or automata from simple ones, following established algorithms. That makes finding suitable patterns of automata less error-prone, which can be very important.

If our language is finite to start with then finding a pattern for it is very easy.

Exercise 33. Show that every finite language is regular.

Assume we have two regular languages, $\mathcal{L}_1$ and $\mathcal{L}_2$. We look at languages we can build from these, and show that these are regular.

Concatenation

In Section 2 it is shown that the concatenation of two regular languages is regular. If $\mathcal{L}_1$ and $\mathcal{L}_2$ are two regular languages, and p_1 is a regular expression defining the first, and p_2 works in the same way for the second, then

$$\mathcal{L}_1 \cdot \mathcal{L}_2 = \mathcal{L}(p_1) \cdot \mathcal{L}(p_2) = \mathcal{L}(p_1 p_2),$$

so this is also a regular language. There is a description how to construct an automaton for $\mathcal{L}_1 \cdot \mathcal{L}_2$ from of those for $\mathcal{L}_1$ and $\mathcal{L}_2$ on page 48.

Kleene star

Again this is something we have already considered. If $\mathcal{L}$ is a regular language then so is $\mathcal{L}^*$. If p is a regular expression for $\mathcal{L}$ then p^* is one for $\mathcal{L}^*$, and again our algorithm for turning patterns into automata shows us how to turn an automaton for $\mathcal{L}$ into one for $\mathcal{L}^*$.

Reversal

If s is a word over some alphabet Σ then we can construct another word over the same alphabet by reading s backwards, or, in other words, reversing it. For a language $\mathcal{L}$ over Σ we define

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

Exercise 34. Here are some exercises concerned with reversing strings that offer a good opportunity for practising what you have learned so far. Parts (a) and (e) require material that is discussed in Appendix A, so it only makes sense to do these if you are working through this part of the notes.

(a) Define the reversal of a string as a recursive function.

(b) Look at an automaton for the language of all non-empty words over the alphabet $\{a, b\}$ which start with a. How can it be turned into one for the language of all words which end with a? *Hint: How could we have a given word take the reverse path through the automaton than it would do ordinarily?*

(c) Look at the language given in Exercise 9 (i). What do the words in its reversal look like? Now look at an automaton for the given language and turn it into one for the reversed language. *Hint: See above.*

(d) In general, describe informally how, given an automaton for a language $\mathcal{L}$, one can draw one for the language $\mathcal{L}^R$.

(e) Take the formal description of a DFA recognizing a language $\mathcal{L}$ as in Definition 5 and turn that into a formal definition for an NFA with ϵ-transitions which recognizes the language $\mathcal{L}^R$.

Unions

If we have regular expressions for $\mathcal{L}_1$ and $\mathcal{L}_2$, say p_1 and p_2 respectively, it is easy to build a regular expression for $\mathcal{L}_1 \cup \mathcal{L}_2$, namely $p_1|p_2$. But how do we build an automaton for $\mathcal{L}_1 \cup \mathcal{L}_2$ from those for $\mathcal{L}_1$ and $\mathcal{L}_2$? We have already seen how to do that as well—form an NFA with a new start state which has ϵ-transitions to the (old) start states of the two automata, as illustrated on page 48. If we like we can then turn this NFA with ϵ-transitions into a DFA.

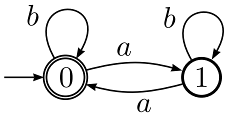
Intersections

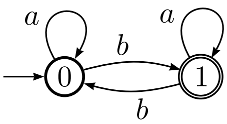
This isn't so easy. It's not at all clear how one would go about about it using patterns, whereas with automata one can see how it might work. The problem is that we can't say 'first get through the automaton for $\mathcal{L}_1$, then through that for $\mathcal{L}_2$': When we have followed the word through the first automaton it has been consumed, because we forget about the letters once we have followed a transition for them. So somehow we have to find a way to let the word follow a path through *both automata at the same time.*

Let's try this with an example: Assume we want to describe the language $\mathcal{L}$ of all words that have an even number of as and an odd number of bs. Clearly

$$\begin{aligned}\mathcal{L} \quad = \quad & \{s \in \{a,b\}^* \mid s \text{ has an even number of } a\text{s}\} \\ & \cap \{s \in \{a,b\}^* \mid s \text{ has an odd number of } b\text{s}\}.\end{aligned}$$

DFAs for those two languages are easy to construct.

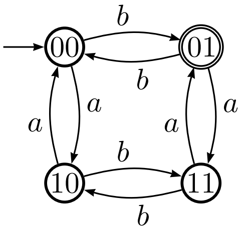




So how do we get an automaton that checks for both properties at the same time? Well, we think of the first automaton as telling us whether the number of as is even so far, or odd, and similarly for the second automaton, and the number of bs. What we need to do in the automaton we want to construct then is to keep track of the as and bs at the same time. So we need four states:

$$(a\text{s even}, b\text{s even}), (a\text{s even}, b\text{s odd}), (a\text{s odd}, b\text{s even}), (a\text{s odd}, b\text{s odd}).$$

The automaton we want looks as follows:



What have we done? We have formed *pairs* of states, a state from the first automaton and a state from the second one. We have then added transitions that literally do follow both automata at the same time.

In general, given DFAs $(Q_1, q^1_\bullet, F_1, \delta_1)$ and $(Q_2, q^2_\bullet, F_2, \delta_2)$ we can form an automaton that recognizes the intersection of the languages recognized by the two DFAs as follows.

- States: $Q_1 \times Q_2$.
- Start state: $(q^1_\bullet, q^2_\bullet)$.
- Accepting states: $F_1 \times F_2$.
- Transition function: δ maps (q_1, q_2) and x to $(\delta(q_1, x), \delta(q_2, x))$. In other words, there is a transition

  $$(q_1, q_2) \xrightarrow{x} (q'_1, q'_2)$$

  if and only if there are transitions

  $$q_1 \xrightarrow{x} q'_1 \qquad \text{and} \qquad q_2 \xrightarrow{x} q'_2.$$

We call the result the **product of the two automata**. If we do need a pattern for the intersection we can now apply Algorithm 2 from page 41.

Exercise 35. Use this construction to draw DFAs recognizing the following languages over the alphabet $\{a, b\}$. First identify the two languages whose intersection you want to form, then draw the automata for those languages, then draw one for their intersection.

(a) All non-empty words that begin with a and end with b.

(b) All words that contain at least two as and at most one b.

(c) All words that have even length and contain an even number of as.

(d) All words that have length at least 3 and whose third symbol (if it exists) is a.

Complements

If $\mathcal{L}$ is a language over the alphabet Σ then we may want to consider its complement, that is

$$\Sigma^* - \mathcal{L},$$

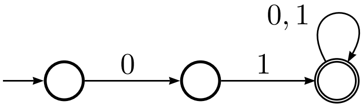
the set of all words over Σ that do *not* belong to $\mathcal{L}$.

Exercise 36. This is a question concerned with the complement of a language.

(a) Consider the language discussed on page 22 of all words which have a 0 in every position that is a multiple of 3. Begin by defining the complement of this language in $\{0,1\}^*$, either using set-theoretic notation or in English.

Now take the DFA for the original language given on page 23 and *turn it into one* for the complement.

(b) Consider the following DFA that recognizes the language of all words over $\{0,1\}$ of length at least 2 whose first letter is a 0 and whose second letter is a 1.



Again, describe the complement of this language either using sets or in English. Then *turn this DFA into one* that recognizes this complement.

(c) Assume you are given a picture that describes a DFA for some arbitrary regular language. Describe in general how you can turn this into a picture for a DFA that recognizes the complement of that language. How does your answer change if you start from an NFA? *Hint: If you find this difficult, then look back at your solutions to (a) and (b). How did you have to change the original DFA?*

(d) Now assume you are given a DFA in the formal description as used in Definition 5, $(Q, q_\bullet, F, \delta)$. Describe how to turn this into a formal description of a DFA recognizing the complement of the original language. (Again you may want to consult Appendix A for this part.)

3.9 Equivalence of Automata

Here is an issue we have not thought about so far. Is there a way of telling whether two automata recognize the same language? You might particularly care about this if you and a friend have created different automata for the same problem and you're not sure whether you're both correct. We say that two automata are **equivalent** if they recognize the same language.Note that if we have an NFA, and we carry out Algorithm 1 for it, then the resulting DFA is equivalent to the NFA we started with.

If you're lucky, this is easy to decide: If you have both drawn a picture of the 'same' automaton, but have merely put different names in your states, then the automata are what we call *isomorphic*. It is easy to show that isomorphic automata recognize the same language.

There are several ways of addressing the more general issue.

Via minimization

Given a DFA it is possible to *minimize the number of states* it has. There is an algorithm which does this, and this calculation has very nice properties. In particular two DFAs recognize the same language if and only if their minimized automata are isomorphic.

However, the algorithm is non-trivial, and showing that it does what it needs to do is fairly involved. For this reason we do not give any detail here.[12] Note that if you have NFAs you first have to find equivalent DFAs.

Via complementation and intersection

We actually have a method which allows us to calculate whether two automata define the same language. Assume we have automata A and A', which recognize the languages $\mathcal{L}$ and $\mathcal{L}'$ respectively. We are interested in the question whether $\mathcal{L} = \mathcal{L}'$. Note that

$$\begin{array}{lll} \mathcal{L} = \mathcal{L}' & \text{if and only if} & \mathcal{L} \subseteq \mathcal{L}' \text{ and } \mathcal{L}' \subseteq \mathcal{L} \\ & \text{if and only if} & \mathcal{L} \cap (\Sigma^* - \mathcal{L}') = \emptyset \text{ and } \mathcal{L}' \cap (\Sigma^* - \mathcal{L}) = \emptyset. \end{array}$$

But we know how to construct automata for the complement of a language, and we also know how to construct automata for the intersection of two languages, so we can reduce our question to that whether the two automata corresponding to the two languages appearing in the above accept no words at all.

However, note that this is quite a bit of work: We have to

- construct two automata for complements—this is not too much work and does not lead to larger automata and
- we have to construct two intersections—the resulting automata will have a size equal to the product of the sizes of the automata involved.

Hence this procedure is not very practical.

Via simulations

Note that both the methods for deciding equivalence of automata we have mentioned so far work for DFAs, but not for their non-deterministic relations. Clearly the question of whether two NFA recognize the same language is even harder than that for two DFAs. We study a different method for comparing automata.

The idea is this: Assume we have two NFAs, say $A = (Q, q_\bullet, F, \delta)$ and $B = (P, p_\bullet, E, \gamma)$. If we can show that for each state q of A there is an 'analogous' state p of B, then every word accepted by the first automaton

[12] Ask me for recommendations if you would like to read more about this! This algorithm is taught in the third year course on compilers.

should be accepted by the second one, assuming we get our definition of 'analogous' right.

Definition 12. *We say that a relation $\sim$ from Q to P is a **simulation** between automata A and B if and only if*

- $q_\bullet \sim p_\bullet$*;*
- *if $q \sim p$ for some $q \in Q$ and $p \in P$ then*
  - *if $q \xrightarrow{x} q'$ then there exists $p' \in P$ such that $p \xrightarrow{x} p'$ and $q' \sim p'$,*
  - *$q \in F$ implies $p \in E$.*

Why are simulations useful? It is because of the following result.

Proposition 3.7. *If there is a simulation from one NFA to another then every word accepted by the first is accepted by the second. Therefore in this case we have that the language accepted by the first automaton is a subset of the language accepted by the second one.*

Why does this hold? A word $x_0x_1 \cdots x_n$ is accepted by automaton $A = (Q, q_\bullet, F, \delta)$ if and only if if there are states

$$q_0 = q_\bullet, q_1, \ldots, q_n$$

such that for all $0 \leq i < n$, $q_i \xrightarrow{x_i} q_{i+1}$ and such that $q_n \in F$, that is, q_n is an accepting state. Because there is a simulation $\sim$ from A to $B = (P, p_\bullet, E, \gamma)$ we have that $q_\bullet \sim p_\bullet = p_0$ and because $q_0 \xrightarrow{x_0} q_1$ we may find a state p_1 such that $p_0 \xrightarrow{x_0} p_1$ and $q_1 \sim p_1$ and so that if q_1 is accepting then so is p_1. So we may construct, one after the other, states p_0, p_1 up to p_n such that for every p_i it is the case that

- $p_i \xrightarrow{x_i} p_{i+1}$,
- $q_i \sim p_i$ and
- q_i accepting implies p_i accepting.

Since q_n is accepting this is also the case for p_n and so the given word is accepted by B.

For deterministic automata we have an even stronger result:

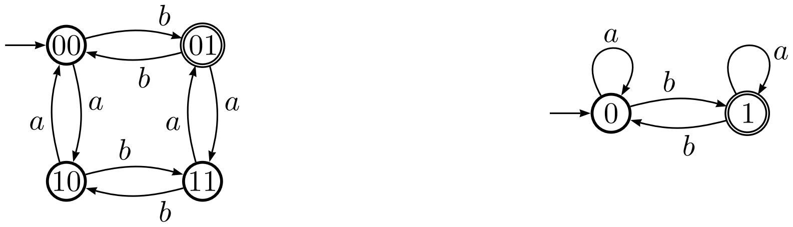
Proposition 3.8. *Let A and B be DFAs. If the language defined by A is a subset of the language defined by B then there exists a simulation from A to B.*

Hence overall we have two theorems, one for NFAs and one for DFAs.

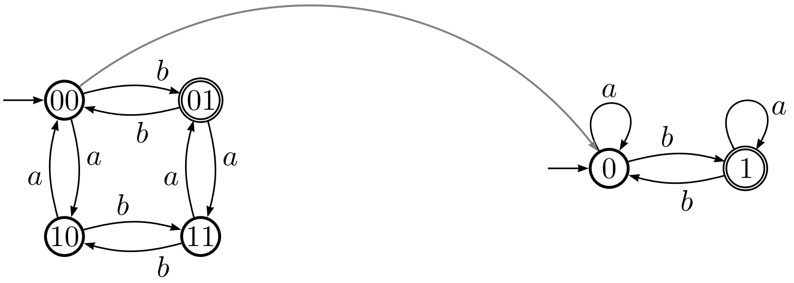
Theorem 3.9. *If A and B are NFAs, and there is a simulation from A to B, as well as one in the opposite direction, then A and B accept the same language.*

Theorem 3.10. *Let A and B be two DFAs. Then A and B accept the same language if and only if there are simulations between them in both directions.*

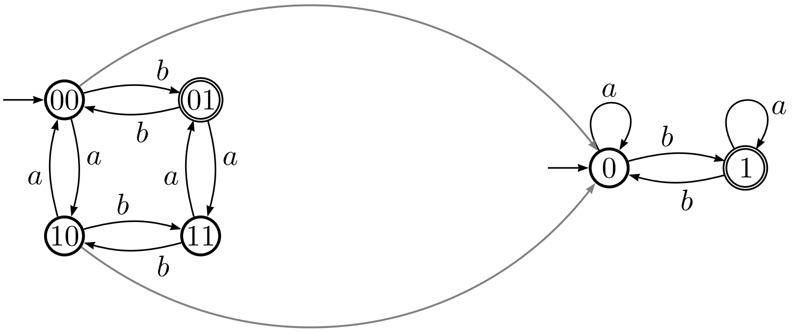
How do we construct simulations in practice? Consider the following automata, assuming we want to construct a simulation from the one on the left to the one on the right.



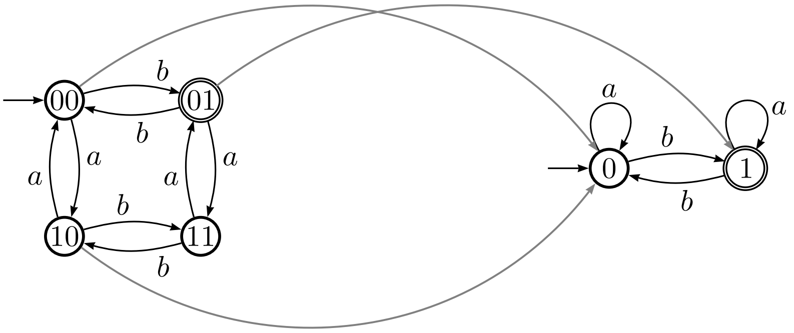
The two initial states have to be in the relation, which we picture as follows:



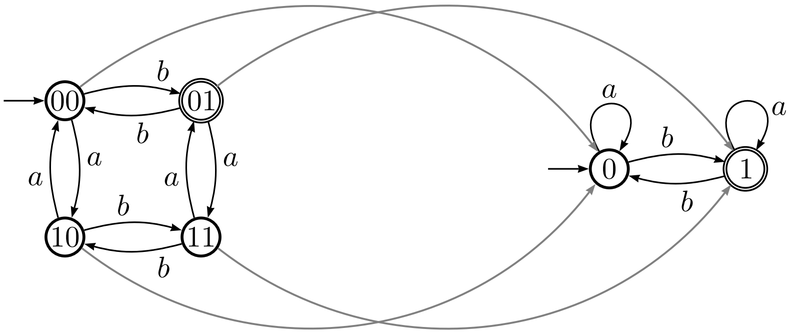
Now the first automaton can make a move $00 \xrightarrow{a} 10$, and we must add find a 'partner' for 10 in the other automaton to match this transition. The only choice is to connect 10 with 0, pictorially:



The first automaton can make a move $00 \xrightarrow{b} 01$, and to match that we have to 'partner' 01 with 1. The state 01 is accepting, but so is 1, so we have satisfied that criterion. We get the following.

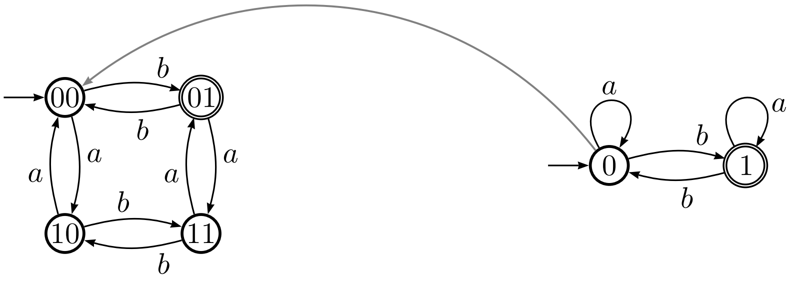


Similarly, we have to find a state matching 11, and that has to be a state which can be reached with b from 10's partner 0, and with a from 01's partner 1, making 1 the only possible choice.

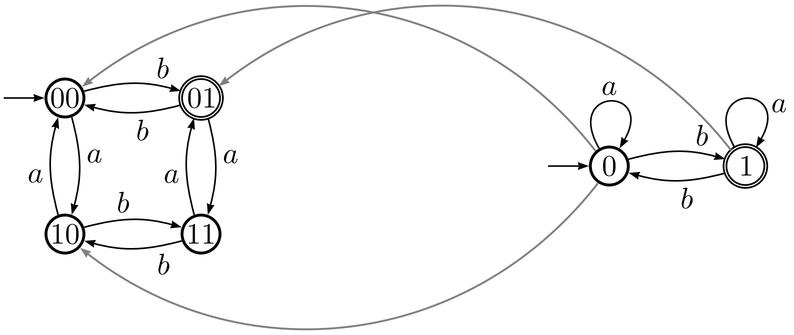


We still have to check that every transition in the automaton on the left can be matched by one between suitable 'partner states' on the right, but this is straightforward.[13]

Now assume we would like to construct a simulation in the *opposite direction*, that is, from right to left. The two start states have to be connected.



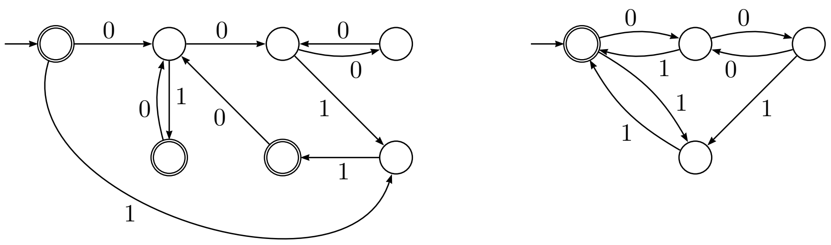
To match transitions from 0 we have to partner the state 0 with 10 and we have to partner the state 1 with 01. The state 1 is an accepting state, but so is the state 01, hence so far we are on track for constructing a simulation.



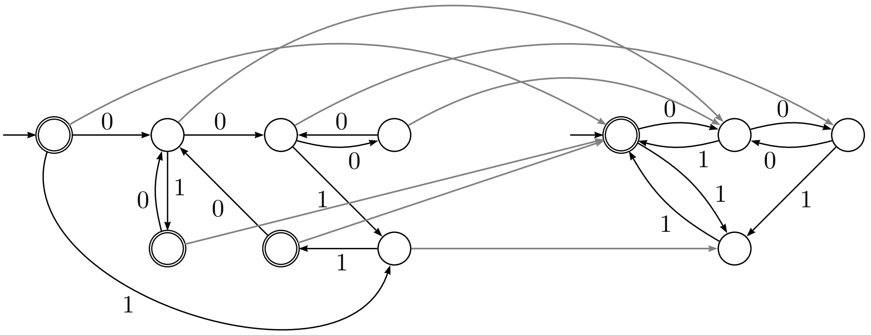
But now we have to match the transition from 1 with a to 1, which would require partnering state 1 with state 11. But 1 is an accepting state and may only be partnered with another accepting state. We had no choice in this construction, and since we cannot complete it to give a simulation, there *does not exist* a simulation from the automaton on the right to the one on the left.

Here is another example. Consider the following two automata.

[13] Note that a state on the left my have more than one partner state on the right—it's coincidence that we do not need more in this example.



We give a simulation from the one on the left to the one on the right in grey (instead of black):

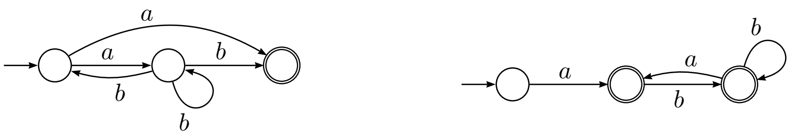


If we invert the simulation so that it goes from the automaton on the right to that on the left then in this case we obtain another simulation. Hence the two automata accept the same language. We call a relation which works in both ways a *bisimulation*.

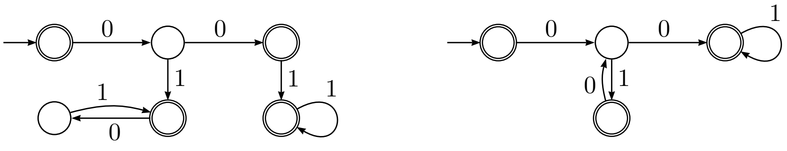
Note that if we have deterministic automata as the target then every simulation my be constructed in this way, one step at a time. One stops when all the states in the source automaton have been matched with a partner in such a way that all transitions in the source automaton can be matched, or when it is impossible to do so. In the first case one has constructed the *unique minimal simulation* from the first automaton to the second, and in the second case one has illustrated that such a simulation does not exist.

Exercise 37. For the following pairs of automata, find simulations going both ways, or argue that one, or both of these, cannot exist.

(a)



(b)



3.10 Limitations of regular languages

So far we have assumed implicitly that all languages of interest are regular, that is, that they can be described using regular expressions or finite automata. This is not really true, but the reason regular languages are so important is that they have nice descriptions, and that they suffice for many purposes.

Something a finite automaton cannot do is to count—or at least, it cannot count beyond a bound defined *a priori*. Such an automaton has a finite number of states, and its only memory is given by those states. Hence it cannot remember information that cannot be encoded into that many states.

If we want an automaton that decides whether or not a word consists of at least three letters then this automaton needs at least four states: One to start in, and three to remember how many letters it has already seen. Similarly, an automaton that is to decide whether a word contains at least 55 as must have at least 56 states.

However, if we try to construct an automaton that decides whether a word contains precisely as many 0s as 1s, then we cannot do this: Clearly the automaton must have a different state for every number of 0s it has already seen, but that would require it to have infinitely many states, which is not allowed.

Similarly, how would one construct a pattern for the language

$$\mathcal{L} = \{0^n 1^n \mid n \in \mathbb{N}\}?$$

We can certainly cope with the language $\{(01)^n \mid n \in \mathbb{N}\}$ by using the pattern $(01)^*$, but that is because once we have seen 0, and the subsequent 1, we may forget about it again (or return to the start state of our automaton). In order to describe $\mathcal{L}$, on the other hand, we really have to remember how many 0s there are before we find the first 1. But there could be any number of 0s, so our automaton would have to have arbitrarily many states—or, if we built a pattern $\boldsymbol{\epsilon}|01|0011|000111|\cdots$ it would be infinite. This is not allowed.

There is a result that allows one to conclude that a particular language is not regular, the **Pumping Lemma**. Looking at that is beyond the scope of this course, but if you really want to understand what this issue you should look this result up. There are automata that do have a memory device which allows them to cope with such languages, but we do not treat them here.

Exercise 38. Which of the following languages are regular? If you answer yes, prove it. If it is no, try to say why.

(a) The language of all words that contain at least as many as as bs.

(b) The language of all strings of brackets that are balanced (so scanning the word from left to right we cannot have more closing brackets at any time than we have seen opening brackets, and for the whole word the number of closing brackets is equal to the number of opening brackets).

(c) The language of all strings over the alphabet $\{0, 1\}$ which, when viewed as a binary number, are divisible by 3.

(d) The language of all strings over the alphabet $\{0, 1\}$ which, when viewed as a binary number, are prime numbers.

(e) The language of all non-empty strings over the alphabet $\{0, 1, \ldots, 9\}$ such that the last symbol is one that occurred earlier in the string.

(f) The language of all strings over $\{0, 1\}$ where the number of 0 is different from the number of 1s.

(g) The language of all strings over $\{a, b\}$ such that each consecutive block of five letters contains at least two as. (Note that these blocks do overlap.)

(h) The language of all palindromes over $\{0, 1\}$, that is, all strings that read forwards as backwards or, to put it differently, the strings which are equal to their reversal.

(i) The language of all strings over $\{a, b\}$ such that there are two 0s in the string which are separated by a number of characters that is a multiple of 4.

(j) The language of all odd-length palindromes over $\{a, b, c\}$.

(k) The language of all words over the alphabet $\{a, b\}$ that have the same number of occurrences of the substring ab as that of the substring ba. The word bab does belong to that language, since it has one occurrence of the substring ab ($[ab]a$) and one of the substring ba ($a[ba]$). *Does your answer change when the alphabet is changed to $\{a, b, c\}$?

(l) The set of all words of the form $s1^n$, where s is a word over $\{a, b, c\}$, and n is the number of letters in s.

3.11 Summary

- We can use *finite automata* to describe the same languages as those given by regular expressions, that is, the regular languages.
- Automata can be *deterministic* (DFA) or *non-deterministic* (NFA). Every DFA can be viewed as an example of an NFA, and for every NFA we can construct a DFA that recognizes precisely the same words.
- Given a DFA we can construct a regular expression such that the language defined by this expression is precisely the language recognized by the automaton we started with.
- Given a regular expression we can construct an *NFA with ϵ-moves* such that the words accepted by the automaton are precisely those that match the regular expression. We can then eliminate the ϵ-moves from the NFA, and use the algorithm mentioned above to turn that into a DFA that recognizes the same language.
- As a result, what we can express using regular expressions, or DFAs, or NFAs, are precisely the same languages.
- We can form the union, intersection, complement, reversal, concatenation and Kleene star operation on regular languages and we will then get another regular languages.
- Not all languages are regular. In particular those languages for which an automaton would have to count up to a boundless number are not regular.

Chapter 4

Describing more complicated languages

At the end of Section 3 it is demonstrated that regular expressions are not sufficient to describe all the languages we are interested in. In particular we cannot even generate the language of all balanced brackets—clearly that is something we have to do when trying to cope with programming languages. How is a compiler to break up a Java program if it cannot cope with checking whether the braces {, } line up properly?

In this section we introduce a way of talking about languages that is more general. However, it is also more complicated, so whenever possible it is a good idea to stick with regular languages.

4.1 Generating words

Instead of thinking of patterns that may by matched by strings it is our aim here to *generate* all the words in our language following some well-defined rules. The way of reading the rules is to see them as ways of *rewriting the string generated so far*, by using one of the rules given to to change one symbol into a string of others. These rules are given by what we call a *grammar*.

Consider the following example:

$$\begin{array}{lll}
B \to 0 & \qquad & S \to B \\
B \to 1 & & S \to (S) \\
B \to 2 & & S \to S + S \\
B \to 3 & & S \to S - S \\
B \to 4 & & S \to S \times S \\
B \to 5 & & S \to S/S \\
B \to 6 & & \\
B \to 7 & & \\
B \to 8 & & \\
B \to 9 & &
\end{array}$$

We have to know where to start, and we always use S as the only symbol we may create from nothing. Once we have S, we can use any of the rules on the right hand side to replace it by a more complicated string, for example

$$\begin{aligned}
S \Rightarrow\ & S \times S \Rightarrow S \times (S) \Rightarrow S \times (S - S) \Rightarrow B \times (S - S) \Rightarrow 5 \times (S - S) \\
& \Rightarrow 5 \times (B - S) \Rightarrow 5 \times (B - B) \Rightarrow 5 \times (3 - B) \ \Rightarrow 5 \times (3 - 4).
\end{aligned}$$

We call this a **derivation** from the grammar.

Exercise 39. Write out derivations for $3/4 + 5$, and 9. Can you write a derivation of 12? Write out a different derivation for $3/4 + 5$. How many of these can you see?

In other words, a grammar is a set of rules that allows us to build certain expressions in a purely mechanical manner. We could program these rules into a computer and ask it to begin to generate strings from it, but unless we were careful regarding what exactly we told it it might build one very long string until it ran out of memory.

Note that the only words we are really interested in are those which consist entirely of the digits, and the arithmetic operators—they no longer contain either S or B. We merely need these two symbols to help us build the string in question: They can be seen as placeholders for longer expressions that haven't yet been built. We call the letters we want our final strings to be built of *terminal symbols* (because they are the only ones there at the end). Once we have introduced a terminal symbol it cannot be replaced by another. The other symbols are called *non-terminal symbols* (because they are not terminal). They are also sometimes called *variables.* Think of them as auxiliary symbols, or placeholders: Once they have done their job they disappear. The rules for the grammar only apply to non-terminal symbols—there can never be a rule that allows us to rewrite a terminal symbol.

Exercise 40. Try to write down a grammar with terminal symbols $\{0, 1\}$ for all non-empty words that start with 0 and end with 1. Show how your grammar generates the strings 01 and 001 and convince yourself that it does not allow you to generate invalid strings such as 11 or 00. *Hint: If you find that difficult at this stage then wait until you have seen the formal definition for a grammar and another example or two.*

We find the above description of a grammar a bit lengthy, and often use an abbreviated form of giving the rules that tell us how we may rewrite a string. For the example this looks as follows.

$$\begin{array}{lcl} B & \to & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ S & \to & B \mid (S) \mid S + S \mid S - S \mid S \times S \mid S/S \end{array}$$

What we have done is to group together all the ways in which we may replace either of the two symbols B and S.

4.2 Context-free grammars

It is time for a formal definition.

Definition 13. *A* ***context-free grammar*** *or* ***CFG*** *is given by the following:*

- *an alphabet* Σ *of* terminal symbols*, also called the* object alphabet*;*
- *an alphabet* Ξ *of* non-terminal symbols*, the elements of which are also referred to as* auxiliary characters*,* placeholders *or, in some books,* variables*, with* $\Xi \cap \Sigma = \emptyset$*;*

- *a special non-terminal symbol $S \in \Xi$ called the* start symbol*;*
- *a finite set of* production rules *of the form $R \rightarrow X$ where $R \in \Xi$ is a non-terminal symbol and $X \in (\Sigma \cup \Xi)^*$ is an arbitrary string of terminal and non-terminal symbols, which can be read as 'R can be replaced by X'.*[1]

In the example above, the object alphabet is

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, \times, /, -, (,)\}.$$

The non-terminal symbols are B and S.

Why do we call grammars whose production rules satisfy these conditions *context-free*? It is because the rules tell us that we may replace any occurrence of a non-terminal as directed. A *context-sensitive grammar* would be able to have rules such as 'You may replace the non-terminal A by the string (A) provided that there is an S immediately to the left of A', so a production rule might look like this: $SA \rightarrow S(A)$.

In general, what context-sensitivity means is that according to their production rules strings consisting of terminal and non-terminal symbols can be replaced by other such strings. The advantage of context-freeness is that we know that once a terminal character is in place, it will remain in the word even if further reduction rules are applied. Further, we don't have to worry about having to choose between different production rules taking us in different directions to the same degree.

Assume we had rules

$$AB \rightarrow A \qquad \text{and} \qquad AB \rightarrow B.$$

Given AB occurring in some string we would then have to make a choice whether to keep the A or the B. For context-free grammars, if we have two distinct non-terminal symbols (such as A and B here) we can apply any one rule for A, and any one rule for B, and it does not matter which order we do that in.

If we allowed context-sensitive rules then our grammars would be much more complicated, and while it would be possible to express more complicated features it would be much harder to see, for example, which language a grammar generates. Natural languages certainly require context-sensitive grammars, but when it comes to programming we can go a very long way without having to use this idea.

The following definition says how we may generate strings from a grammar, and how a grammar defines a language.

Definition 14. *A string $X \in (\Sigma \cup \Xi)^*$* ***is a string generated by grammar*** *G if there is a sequence of strings*

$$S = X_0 \Rightarrow X_1 \Rightarrow \cdots \Rightarrow X_n = X$$

such that each step $X_i \Rightarrow X_{i+1}$ is obtained by the application of one of G's production rules to a non-terminal occurring in X_i as follows.

Let $R \in \Xi$ occur in X_i and assume that there is a production rule $R \rightarrow Y$. Then X_{i+1} is obtained from X_i by replacing one occurrence of R in X_i by the string Y.

The ***language generated by grammar*** *G is the set of all strings over Σ which are generated by G.*[2]

[1] This assumes that we have a special symbol, namely → which should not be contained in either Σ or Ξ to avoid confusion.

[2] Note that such a string may not contain any non-terminal symbols.

Let us look at another example. Let us assume that we want to generate a grammar for the language of all strings that are non-empty and start with 0 over the alphabet $\{0, 1\}$. How do we go about it? Well, we know what our underlying alphabet of terminal symbols Σ is going to be. How about non-terminal ones? This is harder to decide *a priori.* We know that we need at least one of these, namely the start symbols S. Whether or not further non-terminal symbols are required depends on what it is that we have to remember when creating a word (just as the number of states required for a finite automaton depends on what we need to remember about a word[3]).

How do we make sure that our word starts with 0? Well, we have a production rule for S that says $S \to 0\cdots$ then we certainly create 0 as the left-most symbol, and this will stay the left-most symbol. But what should the dots be?

If we try $S \to 0S$ then we do get a problem: Eventually we have to allow 1s into our string, but if we have a rule that says $S \to 1S$ then we could generate words whose first letter is 1. An obvious solution to this problem is to use a new non-terminal symbol, say A. Then we can have a rule

$$S \to 0A,$$

and all we now have to do is to allow A to create whatever letters it likes, one at a time, or to vanish into the empty string. Hence we want rules

$$A \to 0A \mid 1A \mid \epsilon.$$

Exercise 41. Can you create a CFG for the same language using just one non-terminal symbol? *Hint: If you find this tough now then come back to it after you have had more practice.*

Exercise 42. This exercise asks you to create CFGs for various languages over the alphabet $\{0, 1\}$.

(a) All words that contain two consecutive 1s. Now give a derivation for the word 0111. Can you create a grammar for this language with just one non-terminal symbol?

(b) All words that contain at least two 1s. Now give a derivation for 01010.

(c) All words whose length is precisely 3.

(d) All words whose length is at most 3.

(e) All words whose length is at least 3.

(f) All words whose length is at least 2 and whose last but one symbol is 0. Now give a derivation for 0101.

(g) All words for which every letter at an even position is 0. Give a derivation of 101, and of ϵ.

(h) All words consisting of an arbitrary number of 0s followed by the same number of 1s. Try to explain how to generate a word of the form $0^n 1^n$ where $n \in \mathbb{N}$.

(i) All words that contain an even number of 0s. Make sure that your language can generate strings such as 001010 and 0001001000.

[3] However, in a finite state machine we only get to scan the word from left to right; when creating a derivation we may be working on several places at the same time.

(j) All words that do not contain the string 01.

Which ones of those are easy to do, which ones harder? Look at other languages from Exercises 9 and 10. Which ones of those, do you think, would be particularly hard?

Part (h) of this exercise already indicates that we can express languages using CFGs that we cannot express using finite automata or regular expressions.

We give another example. Assume we want to generate the set of all words of the language

$$\{s1^n \mid s \in \{a, b, c\}^*, n \in \mathbb{N} \text{ and } n \text{ is the number of letters in } s\}.$$

This cannot be a regular language since it would require an automaton to remember how many letters it has seen, and that number might rise above any bound (compare Exercise 38.

Our grammar has to be such that every letter is matched by 1, so we have to create them simultaneously. If we use $\Sigma = \{a, b, c, 1\}$ and $\Xi = \{S\}$ the following production rules will do just that.

$$S \to aS1 \mid bS1 \mid cS1 \mid \epsilon.$$

Here is another exercise that asks you to find grammars for other non-regular languages (compare this with Exercise 38).

Exercise 43. Give CFGs for the following languages.

(a) The language of all strings of brackets that are balanced (so we cannot have more closing brackets at any time than we have seen opening brackets, and for the whole word the number of closing brackets is equal to the number of opening brackets).

(b) The language of all palindromes over $\{a, b, c\}$.

(c) The language of all odd-length palindromes over $\{a, b, c\}$ whose middle letter is a.

(d) The language

$$\{a^i b^j c^k \mid i, j, k \in \mathbb{N}, \, k = i + j\}.$$

We have a name for the languages that can be described by a context-free grammar.

Definition 15. *A language $\mathcal{L}$ over Σ is **context-free** if there is a context-free grammar whose alphabet of terminal symbols is Σ such that $\mathcal{L}$ is generated by the grammar*

Exercise 44.* Consider the following grammar. $\Sigma = \{a, b\}$, $\Xi = \{S\}$, $S \to SaSaSbS \mid SaSbSaS \mid SbSaSaS \mid \epsilon$. Argue that this CFG generates the language of all words that contain precisely twice as many as as bs.

Exercise 45. Design a CFG for the language consisting of all words that contain precisely as many as as bs. Try to argue that you really can generate all such words using your production rules. *Hint: You could analyse the language from the previous exercise. Alternatively, try to come up with a way that keeps track of the number of as or bs generated so far.*

4.3 Regular languages are context-free

We have already seen that we can capture non-regular languages using context-free grammars. It may seem fairly clear that CFGs are more powerful than regular expressions, but how do we know that we can express *every* regular language using a CFG?

This turns out not to be so very difficult. Maybe surprisingly it is easier here to start with a DFA than with a regular expression.

Assume that we have a regular language $\mathcal{L}$ over some alphabet Σ. By Theorem 3.5 we can find a DFA, say $(Q, q_\bullet, F, \delta)$, that recognizes this language.

We now define a CFG from the automaton as follows:

- We use Σ as the alphabet of terminal symbols.
- We use $\Xi = Q$ as the alphabet of non-terminal symbols.
- We use $S = q_\bullet$ as the start symbol.
- For every transition

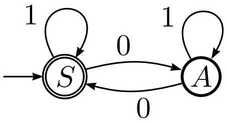
  $$q \xrightarrow{x} q'$$

  in the automaton (that is, for every pair $(q, x) \in Q \times \Sigma$ and $q' = \delta(q, x)$) we introduce a production rule

  $$q \rightarrow xq'.$$

- For every accepting state $q \in F$ we add a production rule $q \rightarrow \epsilon$.

We recall the following automaton from Section 3 where we have changed the names of the states.



We can now use non-terminal symbols S and A as well as terminal symbols 0 and 1 for a CFG with the following production rules:

$$\begin{array}{lcl} S & \rightarrow & \epsilon \mid 1S \mid 0A \\ A & \rightarrow & 1A \mid 0S \end{array}$$

This grammar will generate the language of those words consisting of 0s and 1s which contain an even number of 0s.

Exercise 46. Use this construction of a context-free grammar to produce grammars for the regular languages defined as follows:

(a) By the pattern $(ab|a)^*$. *Hint you should already have a DFA for this pattern from Exercise 32.*

(b) $(01)^*|001^*$ *Hint: as above.*

(c) The words that do not contain the string 010. *Hint: This language appears in Exercises 10 and 14 so you already have a suitable DFA.* Could you have found such a grammar without using this method?

(d) By the pattern $ab^*c^*|a^*b^*c$. Could you have found such a grammar without using this method?

We note that the production rules we have used here are very limited: for an arbitrary non-terminal symbol $R \in \Xi$ they are of the form $R \to xR'$, where $x \in \Sigma$ and $R' \in \Xi$, or $R \to x$, where[4] $x \in \Sigma$, or $R \to \epsilon$. We call grammars were all production rules are of this shape **right-linear**. Such grammars are particularly simple, and in fact every language generated by a right-linear grammar is regular.

We can do even more for regular languages using CFGs. This is our first truly applied example of a context-free grammar.

There is a context-free grammar for the language of regular expressions over some alphabet Σ.

The underlying alphabet of terminal symbols[5] we require here is

$$\Sigma \cup \{\boldsymbol{\epsilon}, \boldsymbol{\emptyset}, |, *\}.$$

The alphabet of non-terminal symbols can be $\{S\}$. We use the following production rules:

- $S \to \boldsymbol{\emptyset}$.
- $S \to \boldsymbol{\epsilon}$.
- $S \to x$, for all $x \in \Sigma$.
- $S \to SS$ for concatenation.
- $S \to S|S$ for alternative.
- $S \to S^*$ for the Kleene star.

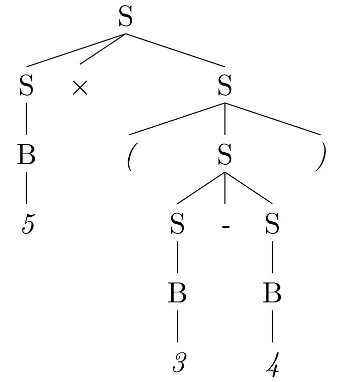
4.4 Parsing and ambiguity

When a compiler deals with a piece of code it has to *parse* it. In other words, it has to break it down into its constituent parts in a way that makes sense. Similarly, if you were to write a program that could take (possibly quite long) arithmetic expressions as input and evaluate it, it would have to break down the expression to decide in which order to carry out the various operations.

When we give a derivation we automatically provide one way of breaking down the given string. Parsing is an attempt of carrying out the *opposite* process, namely to take a string and find out how to assemble it from simpler parts. Instead of finding a derivation it is often more meaningful to create a **parse tree** which gives a better overview of how the various bits fit together.

[4] We did not need that rule above, but we need it to be part of this definition.

[5] It would not be a good idea to call this alphabet Σ this time.

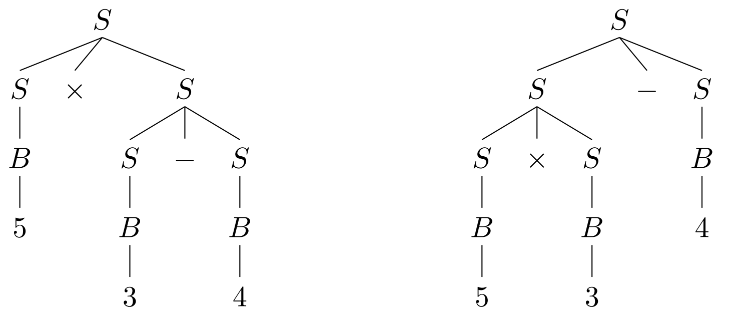


We can read off the string generated from the leaves of the tree,[6] here it is 5 × (3 − 4), and we can also see which production rules to use to get this result. If we compare this with a derivation like that given on page 63 we can see that in general, there will be several derivations for every parse tree. This is because the parse tree does not specify in which order the rewriting of non-terminal symbols should be carried out.

If we have $S \times S$, should we first replace the left S by B, or the right one by (S)? The answer is that it doesn't really matter, since we can do this in either order, and the strings we can get to from there are just the same. What the parse tree does then is to remember the *important* parts of a derivation, namely which rules to apply to which non-terminals, while not caring about the order some of these rules are applied.

Any non-trivial word in a context-free language typically has many different derivations. However, ideally we would like every word to only have one parse tree for reasons explained below.

Consider the string 5 × 3 − 4. This has the following two parse trees.



Now if a computer (or a calculator) is to decide how to calculate this number it has to break down the string. The parse trees above say that this can be done in two ways, which we might write as $(5 \times 3) - 4$ and $5 \times (3 - 4)$. The first is calculated as $15 - 4 = 11$, while the second gives $5 \times (-1) = -5$, so they really do give rise to quite different things.

Of course we have generally adopted the convention that it is the first expression that we mean when we write $5 \times 3 - 4$, because the convention says that we should multiply before we add. But if we look at our grammar then there is nothing that specifies that a computer should treat this expression in this way.

If there are two different parse trees for the same word we say that it is **ambiguously generated** and we say that the grammar is **ambiguous**. When designing grammars for languages that require that the generated

[6] Note that ϵ may appear as a leaf, in which case we ignore it from reading off the word in question.

words are evaluated in some way then one should always aim to give a *unambiguous* grammar. However, it is not always possible to do this, and if that is the case then we call the language in question **inherently ambiguous**.

We can change the grammar given above in such a way that every word it generates can be parsed unambiguously as follows.

$$\begin{array}{lcl} B & \to & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ S & \to & B \mid (S+S) \mid (S-S) \mid (S \times S) \mid (S/S) \end{array}$$

Now when we derive a word we automatically say in which order the various operations should be carried out because there are brackets doing this for us.[7] Note that a word still has a number of different *derivations*, it is the number of *parse trees* that should be equal to one.

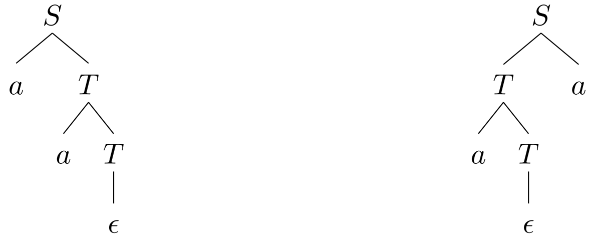
The first thing a compiler has to do when given a program is to recognize various key words, which are often called *tokens*. Examples for tokens for the `Java` programming languages are words such as `if`, `else`, `public`, `class`, and many others, but also the names of variables and methods. This is referred to as a *lexical analysis*, and this is typically carried out using regular expressions. The Unix command `lex` generates such analysers.

Once this is done the compiler has to parse the program—in other words, it produces a parse tree for the program. If that cannot be done then the compiler will raise a syntax error. There are some standard tools to help with this kind of thing: The Unix `YACC` command takes a context-free grammar and produces a parser for that grammar. Clearly for a program it is particularly important that it can be uniquely parsed so that there is precisely one way for the computer to carry out the constructions contained in it. Because programmers do not like to put in a lot of brackets typically a compiler will have a list of which operations should be carried out first.

We look at another example of ambiguity. Consider the following grammar.

$$\begin{array}{lcl} S & \to & aT \mid Ta \\ T & \to & aT \mid bT \mid \epsilon \end{array}$$

The word aa has two parse trees for this grammar:



Hence we know that this grammar is also ambiguous. How can we turn it into an unambiguous one? First of all we have to find out which language is described by the grammar. In this case this is not so difficult: It is the set of all words over the alphabet $\{a, b\}$ which start with a or end with a (or both). The ambiguity arises from the first rule: The a that is being

[7] The usual convention that tells us that multiplication should be carried out before addition is there so that we do not have to write so many brackets for an expression while it can still be evaluated in a way that leads to a unique result.

demanded may either be the first, or the last, letter of the word, and if the word starts and ends with a then there will be two ways of generating it.

What we have to do to give an unambiguous grammar for the same language is to pick the first or last symbol to have priority, and the other one only being relevant if the chosen one has not been matched. A suitable grammar is as follows:

$$\begin{array}{lcl} S & \to & aT \mid bU \\ T & \to & aT \mid bT \mid \epsilon \\ U & \to & aU \mid bU \mid a \end{array}$$

Now we generate any word from left to right, ensuring the grammar is unambiguous. We also remember whether the first symbol generated is a, in which case the remainder of the word can be formed without restrictions, or b, in which case the last symbol has to be a. It turns out that we do not need three non-terminal symbols to do this: Can you find a grammar that uses just two and is still unambiguous?

Exercise 47. For both grammars over the alphabet $\Sigma = \{a, b\}$ given below do the following: Show that the grammar is ambiguous by finding a word that has two parse trees, and give both the parse trees. Now try to determine the language generated by the grammar. Finally give an unambiguous grammar for the same language.

(a) Let $\Xi = \{S, T\}$ with production rules $S \to TaT$ and $T \to aT \mid bT \mid \epsilon$.

(b) The grammar with $\Xi = \{S\}$ and $S \to aS \mid aSbS \mid \epsilon$.

4.5 A programming language

Defining a programming language is a huge task: First we have to describe how programs can be formed (we have to define the **syntax** of the language), and then we have to describe how a computer should act when given a particular program (we have to give a **semantics**).

The syntax of a language tells us how to write programs that are considered well-formed (so if it is a compiled language, for example, the compiler will not give any syntax errors). However, this says nothing about what any program actually *means*. When you see a fragment of a program you probably have some kind of idea what at least some of the instructions within it are supposed to do, but that is because program designers stick to certain conventions (so that $a + b$ will typically mean adding the content of the variable a to that of the variable b), but a computer doesn't know anything about these preconceptions. Therefore in order to fully define a programming language one also has to say what a program actually means. There is a third year course unit that concerns itself with different ways of doing so.

Most real programming languages such as Java, C, or ML, or PHP, typically have very large grammars and we do not have the time here to study such a big example.

We look at a very small programming language called While. At first sight it looks as if there can't possibly anything interesting to be computable with it: It only knows about boolean data types (such as true, T, here `tt`, and false, F, here `ff`), and natural numbers, say from 0 to some number we call `maxint`. We also need variables which can take on the value of any of our natural numbers. Let's assume for the moment we only have three variables, x, y, and z. The language has three kinds of entities:

- arithmetic expressions A
- boolean expressions B and
- statements S.

For each of these we have a production rule in our grammar.

$$\begin{array}{rcl} A & \rightarrow & 0 \mid 1 \mid \cdots \mid \texttt{maxint} \mid x \mid y \mid z \mid A+A \mid A \times A \mid A - A \\ B & \rightarrow & \texttt{tt} \mid \texttt{ff} \mid A = A \mid A \leq A \mid B \wedge B \mid \neg B \\ S & \rightarrow & x := A \mid y := A \mid z := A \mid \texttt{skip} \mid S; S \mid \texttt{if}\, B\, \texttt{then}\, S\, \texttt{else}\, S \mid \texttt{while}\, B\, \texttt{do}\, S \end{array}$$

This may surprise you, but if we add more variables to our language[8] then it can calculate precisely what, say, Java can calculate for natural numbers and booleans.

We can now look at programs in this language. Assume that the value held in the variable y is 10.

$$x := 1; \texttt{while}\ \neg y = 0\ \texttt{do}\, x := 2 \times x; y := y - 1$$

This program will calculate 2^{10} and store it in x. In fact, it will calculate 2^y (assuming y holds a natural number).

Exercise 48. We look at this example in more detail.

(a) Give a derivation for the program above.

(b) Is this grammar unambiguous? If you think it isn't, then give an example and show how this grammar might be made unambiguous, otherwise explain why you think it is.

(c) Give a parse tree for the above program. Can you see how that tells you something about what computation is supposed to be carried out?

4.6 The Backus-Naur form

In the literature (in particular that devoted to programming languages) context-free grammars are often described in a slightly different form to the one used here so far. This is known as the *Backus-Naur form* (sometimes also described as *Backus Normal Form* but that isn't a very accurate name). There is a simple translation process.

- Non-terminal symbols don't have to be capitals and are distinguished from other symbols by being in *italics*, or they are replaced by a descriptive term included in angled brackets ⟨ and ⟩.
- The rewriting arrow → is replaced by the symbol ::=.

[8] When people define grammars for programming languages they typically use abbreviations that allow them to stipulate 'any variable' without listing them all explicitly, but we don't want to introduce more notation at this stage.

We repeat the grammar for the While language from above to illustrate what this looks like.

Here we assume that a ranges over arithmetic expressions, **AExp**, b ranges over boolean expressions, **BExp**, and S ranges over statements, **Stat**.

$$\begin{array}{rcl}
a & ::= & 0 \mid 1 \mid \cdots \mid \mathtt{maxint} \mid x \mid y \mid z \mid a+a \mid a\times a \mid a-a \\
b & ::= & \mathtt{tt} \mid \mathtt{ff} \mid a=a \mid a\leq a \mid b\wedge b \mid \neg b \\
S & ::= & x:=a \mid y:=a \mid z:=a \mid \mathtt{skip} \mid S;S \mid \mathtt{if}\,b\,\mathtt{then}\,S\,\mathtt{else}\,S \mid \mathtt{while}\,b\,\mathtt{do}\,S
\end{array}$$

In the next example we assume that instead we are using $\langle\text{aexp}\rangle$ to range over **AExp**, $\langle\text{bexp}\rangle$ over **BExp**, and $\langle\text{stat}\rangle$ over **Stat** respectively.

$$\begin{array}{rcl}
\langle\text{aexp}\rangle & ::= & 0 \mid 1 \mid \cdots \mid \mathtt{maxint} \mid x \mid y \mid z \mid \\
 & & \langle\text{aexp}\rangle + \langle\text{aexp}\rangle \mid \langle\text{aexp}\rangle \times \langle\text{aexp}\rangle \mid \langle\text{aexp}\rangle - \langle\text{aexp}\rangle \\
\langle\text{bexp}\rangle & ::= & \mathtt{tt} \mid \mathtt{ff} \mid \langle\text{aexp}\rangle = \langle\text{aexp}\rangle \mid \langle\text{aexp}\rangle \leq \langle\text{aexp}\rangle \mid \\
 & & \langle\text{bexp}\rangle \wedge \langle\text{bexp}\rangle \mid \neg\langle\text{bexp}\rangle \\
\langle\text{stat}\rangle & ::= & x := \langle\text{aexp}\rangle \mid y := \langle\text{aexp}\rangle \mid z := \langle\text{aexp}\rangle \mid \mathtt{skip} \mid \langle\text{stat}\rangle;\langle\text{stat}\rangle \mid \\
 & & \mathtt{if}\;\langle\text{bexp}\rangle\;\mathtt{then}\;\langle\text{stat}\rangle\;\mathtt{else}\;\langle\text{stat}\rangle \mid \mathtt{while}\;\langle\text{bexp}\rangle\;\mathtt{do}\;\langle\text{stat}\rangle
\end{array}$$

4.7 Properties of context-free languages

There is a description for building new regular languages from old ones in Section 3. We can use almost any way we like to do so; set-theoretic ones such as unions, intersections and complements, as well as ones using concatenation or the Kleene star; even reversing all the words work. Context-free languages are rather more fiddly to deal with: Not all of these operations work for them.

Concatenation. This does work. We do the following. Assume that we have two grammars with terminal symbols taken from the alphabet Σ, and non-terminal symbols Ξ_1 respective Ξ_2. We now take every symbol in Ξ_1 and put a subscript 1 onto it, and similarly for every symbol in Ξ_2, and the subscript 2. We have now forced those two alphabets to be disjoint, so when we form their union the number of symbols in the union is the sum of the symbols in Ξ_1 and Ξ_2. We add a new start symbol S to the set of non-terminal symbols. We now take all the production rules from the first grammar, and put subscripts of 1 onto each non-terminal symbol occurring in it, and do the same for the second grammar with the subscript 2. We add one new production rule $S \to S_1S_2$.

Exercise 49. Use your solution to Exercise 42 (c) to produce a grammar for the language of all words whose length is at least 6, using this procedure.

Kleene star. This looks as if it should be more complicated than concatenation, but it is not. We merely add two production rules to the grammar (if they aren't already there) for our given language, namely $S \to SS$, and $S \to \epsilon$. If we then wish to generate a word that is the n-fold concatenation of words in our language, where $n \in \mathbb{N}^+$ we merely start by applying the rule $S \to SS$ $(n-1)$ times, giving us n copies of S. For each one of these we can then generate the required word. If we wish to generate the empty word we can do this by applying the second new rule, $S \to \epsilon$.

Exercise 50. Use your solution to Exercise 42 (c) to produce a grammar for the language of all words whose length is divisible by 3, using this procedure.

Reversal. This is quite easy. Leave the two alphabets of terminal and non-terminal symbols as they are. Now take each production rule, and replace the string on the right by its reverse. So if there is a production rule of the form $R \to 00R1$, replace it by the rule $R \to 1R00$.

Exercise 51. Use this procedure to turn your solution for Exercise 40 into a grammar that generates all words over $\{0, 1\}$ that start with 1 and end with 0.

Unions. This does work.

Exercise 52. Show that the union of two context-free languages over some alphabet is context-free.

Intersections. Somewhat surprisingly, this is not an operation that works. The intersection of a context-free language with a regular one is context-free, but examining this issue in more detail goes beyond the scope for this course.

Complements. This also does not work. Just because we have a way of generating all the words in a particular set does not mean we can do the same for all the words *not* belonging to this set.

4.8 Limitations of context-free languages

Given the idea of context-sensitive grammars mentioned at the start of this section it should not come as a surprise that there are languages which are not context-free. We do not develop the technical material here that allows us to prove this statement. In Section 3 there is an explanation why finite state automata can only do a very limited amount of counting, and context-free grammars can certainly do more than that. However, there are limitations to this as well. The language

$$\{a^n b^n c^n \mid n \in \mathbb{N}\}$$

over $\{a, b, c\}$ is not context-free.

There is a formal result, called *the Pumping Lemma for context-free languages* that describes some of the possible properties that indicate that a language fails to be context-free.

There are automata which can be used to describe the context-free languages which are known as *pushdown-automata.* These do have a memory device that takes the form of a stack. That makes them fairly complicated to cope with, and that is why they are beyond the scope of this course. Moreover, for these automata, deterministic and non-deterministic versions are not equivalent, making them even more complex to handle.

4.9 Summary

- A *context-free grammar* or *CFG* is given by two alphabets, one of *terminal*, and one of *non-terminal* symbols and by production rules that tell us how to replace non-terminals by strings consisting of both, terminal and non-terminal symbols.

- We generate words for a CFG by starting with the non-terminal start symbol S. In the word generated so far we then pick one non-terminal symbol and replace it by a string according to one of the production rules for that symbol.

- A CFG generates a language made up of all the words consisting entirely of terminal symbols that can be generated using it. The languages that can be generated in this way are known as the *context-free languages*.

- Every regular language is context-free, but there are some context-free languages that are not regular.

- Rather than a derivation for a word we can give a *parse tree* which is sufficient to tell us how to carry out calculations, for example. Programming languages are defined using grammars, and a compiler parses a program to check whether it is correctly formed.

- We can apply the union, concatenation, Kleene star and reversal operations to context-free languages and that gives us another context-free language.

- There are languages that are not context-free.

Glossary

accept

A word can be accepted, or not accepted, by an automaton. 26, 28, 46

Algorithm 1

Turns an NFA into a DFA. 30

Algorithm 2

Turns an automaton (works for NFA and DFA, but more complicated for NFA) into a pattern describing the language recognized by the automaton. 41

Algorithm 3

Takes a pattern and turns it into an NFA with ϵ-transition such that the language recognized by the automaton is that defined by the pattern. 48

Algorithm 4

Turns an NFA with ϵ-transitions into an NFA without these. 46

alphabet

A set of letters. 12

ambiguous

A word can be ambiguously generated if we have a grammar that has two parse trees for it, and if such a word exists we call the grammar ambiguous. A language is inherently ambiguous if all the grammars generating it are ambiguous. 68

Backus-Naur form

A particular way of writing down a grammar popular within computer science, in particular for describing the syntax of programming languages. 71

CFG

Rules for generating words by using rewrite rules known as a grammar, in our case a context-free one. 62

concatenation

An operation on words (or letters) that takes two words and returns one word by attaching the second word to the end of the first word. 12

context-free

A language is context-free if there is a context-free grammar generating it. (Note that grammars can also be context-free—we do not consider other grammars on this course. 65

context-free grammar

Same as CFG. 62

derivation

A way of demonstrating that a particular word is generated by a grammar. 62

deterministic finite automaton

Same as DFA. 25

DFA

An alternative way of defining a language, more suitable for human consumption than a regular expression. 25

ϵ

The empty word. 12

Kleene star

An operation allowing an arbitrary number of concatenations of words from a given language. 14

language

A set of words. 13

language defined by pattern

The set of all the words matching the pattern. See also the definition on page. 18

language generated by grammar

The set of all words consisting entirely of terminal symbols which can be built using the rules of the grammar. 63

letter

Building block for word. Same as symbol. 12

matching

A word can match a pattern. 17

NFA

An alternative way of defining a language, often the easiest for people to devise. 27

NFA with ϵ-transitions

An NFA which also allows transitions labelled with ϵ which do not have to be matched against a letter. 45

non-deterministic finite automaton

Same as NFA. 27

non-terminal symbol

An auxiliary symbols used to describe a grammar—we want to create words that do not contain auxiliary symbols. 62

parse tree

A parse tree shows how a string can be generated from a given grammar in a more useful way than a derivation. 67

pattern

A particular kind of string used to define languages. 16

recognize

Automata recognize languages. 26, 28, 46

regular expression

Same as pattern. 16

right-linear

A grammar is right-linear if its production rules are particularly simple. The definition is given on page. 67

simulation

A special relation between the states of two automata; its existence implies that the language accepted by the first automaton is a subset of that accepted by the second one.. 56

string

Same as word. 12

string generated by grammar

A word consisting of terminal and non-terminal symbols that we can build by using the rules of the grammar. 63

symbol

Same as letter. 12

terminal symbol

An element of the alphabet over which we are trying to create words using a grammar. 62

unambiguous

A grammar is unambiguous if it isn't ambiguous. 69

word

Obtained from concatenating 0 or more letters. Same as string. 12

Appendix A

In the language of mathematics

The account given so far is informal as far as possible. To be completely rigorous the language of mathematics has to be used to define the various notions we have used. For the interested reader we here give a glimpse of how this is done. It is possible to pass this course, and pass it with a good mark, without knowing the content of this chapter, but those who want to get the whole picture, and get a really good mark, should work through this part as well.

A.1 The basic concepts

We begin by giving mathematical definitions for the various notions from Section 2.1.

Definition 16. *An* **alphabet** Σ *is a finite set of characters (primitive indecomposable symbols) called the* **letters** *of* Σ.

Definition 17. *Let* Σ *be an alphabet. A* **word over** Σ *is a finite string*

$$x_1 x_2 \cdots x_n$$

where $n \in \mathbb{N}$ *and the* x_i *are letters from* Σ. *If* $n = 0$ *we write* ϵ *for the resulting string.*

Exercise A.1. When we are given an alphabet we can look at various kinds of words we can build from it.

(a) Form all the three letter words over the alphabet $\{a, b, 1\}$.

(b) Form all the words over the alphabet $\{0\}$.

(c) Form all the words over the alphabet $\{\}$.

We can alternatively use a recursive definition for a word. This is particularly useful if we want to define operations on words recursively. It also fits well with how some data structures are defined.

Alternative definition. *A* **word** *over the alphabet* Σ *is obtained from the following rules:*

- *The empty word* ϵ *is a word.*
- *If* s *is a word and* $x \in \Sigma$ *then* sx *is a word.*

Now that we have the basic concept we can define operations for words, such as the length of a word. Note how this definition follows the recursive definition of a word.

Definition 18. *The* **length** $|s|$ *of a word* s *over some alphabet is defined as follows:*

$$|s| \quad = \quad \begin{cases} 0 & s = \epsilon \\ |s'| + 1 & s = s'x \end{cases}$$

Exercise A.2. Try to come up with a non-recursive definition of the length function. *Hint: Look at the original definition of word, or at the definition of concatenation to get an idea.* Use the recursive definition of a word to argue that your definition agrees with the original.

We have a binary operation on words, namely concatenation.

Definition 19. *Given an alphabet* Σ*, concatenation is an operation from pairs of words to words, all over* Σ*, which, for word* s *and* t *over* Σ*, we write as* $s \cdot t$*. It is defined as follows:*

$$x_1 \dots x_m \cdot y_1 \dots y_n = x_1 \dots x_m y_1 \dots y_n.$$

Exercise A.3. Recall the definition of an associative or commutative operation from COMP11120.

(a) Argue that the concatenation operation is associative.

(b) Show that the concatenation operation is not commutative.

Exercise A.4. Use recursion to give an alternative definition of the concatenation operation. (You may find this difficult—try to give it a go anyway.)

Exercise A.5. Show that $|s \cdot t| = |s| + |t|$. *Hint: You may well find it easier to use the non-recursive definition for everything.*

Exercise A.6. Practice your understanding of recursion by doing the following.

(a) Using the recursive definition of a word give a recursive definition of the following operation. It takes a word, and returns the word where every letter is repeated twice. So *ab* turns into *aabb*, and *aba* into *aabbaa*, and *aa* to *aaaa*.

(b) Now do the same thing for the operation that takes a word and returns the reverse of the word, so *abc* becomes *cba*.

In order to describe words of arbitrary length concisely we have adopted notation such as a^3 for *aaa*. This works in precisely the same way as it does for powers of numbers: By a^3 we mean the word that results from applying the concatenation operation to three copies of a to obtain *aaa*, just as in arithmetic we use 2^3 to indicate that multiplication should be applied to three copies of 2 to obtain $2 \times 2 \times 2$. So all we do by writing a^n is to find a shortcut that tells us how many copies of a we require (namely n many) without having to write them all out.

Because we know both these operations to be associative we do not have to use brackets here: $(2 \cdot 2) \cdot 2$ is the same as $2 \cdot (2 \cdot 2)$ and therefore the notation $2 \cdot 2 \cdot 2$ is unambiguous, just as is the case for $2 \times 2 \times 2$.

What should a^1 mean? Well, this is simple, it is merely the word consisting of one copy of a, that is a. The question of what a^0 might mean is somewhat trickier: What is a word consisting of 0 copies of the letter a? A useful convention in mathematics is to use this to mean the unit for the underlying operation. Hence in arithmetic $2^0 = 1$. The unit for concatenation is the empty word and so we think of a^0 as a way of referring to the empty word ϵ.

This way we obtain useful rules such as $a^m \cdot a^n = a^{n+m}$. Similarly we have $(a^m)^n = a^{nm}$, just as we have for exponentiation in arithmetic. However, note that $(ab)^n$ consists of n copies of ab concatenated with each other, rather than $a^n b^n$, as we would in arithmetic.[1]

Exercise A.7. Write out in full the words 0^5, $0^3 1^3$, $(010)^2$, $(01)^3 0$, 1^0.

Languages are merely collections of words.

Definition 20. *Let Σ be an alphabet. A* **language over** Σ *is a set of words over* Σ.

As mentioned in Chapter 2 using this definition we automatically obtain set-theoretic operations: We can form the unions, intersections, complements and differences of languages in precisely the same way as we do this for other sets. Hence expressions such as $\mathcal{L}_1 \cap \mathcal{L}_2$, $\mathcal{L}_1 \cup \mathcal{L}_2$ and $\mathcal{L}_1 \setminus \mathcal{L}_2$ are immediately meaningful.

Exercise A.8. Let Σ be the alphabet $\{0, 1, 2\}$ and let

$$\mathcal{L}_1 = \{s \mid s \text{ is a word consisting of 0s and 1s only}\},$$

$$\mathcal{L}_2 = \{s \mid s \text{ is a word beginning with 0 and ending with 2}\}.$$

Calculate the following languages: $\mathcal{L}_1 \cap \mathcal{L}_2$, $\mathcal{L}_1 \cup \mathcal{L}_2$ and the complement of $\mathcal{L}_1$ in the language of all words over Σ.

Note that we have notation to find a more compact description of the language

$$\{\epsilon, 1, 11, 111, 1111, \ldots\}$$

of all words over the alphabet $\{1\}$ as

$$\{1^n \mid n \in \mathbb{N}\}.$$

Exercise A.9. Write down the following languages using set-theoretic notation:

(a) All the words consisting of the letters a and b which contain precisely two as and three bs.

(b) All the words consisting of the letter 1 that have even length.

(c) All the words consisting of an arbitrary number of as followed by at least one, but possibly more, bs.

(d) All the non-empty words consisting of a and b occurring alternatingly, beginning with an a and ending with a b.

[1] The reason this rule doesn't hold is that the concatenation operation isn't commutative (see Exercise A.3‘), and so we can't swap over the as and bs to change the order in which they appear.

(e) All the non-empty words consisting of a and b occurring alternatingly, beginning with an a and ending with an a.

(f) All the words consisting of an arbitrary number of 0s followed by the same number of 1s.

(g) All the words over some alphabet Σ.

Note that if we have two alphabets $\Sigma' \subseteq \Sigma$ then every word over Σ' can be viewed as a word over the alphabet Σ. Alternatively, we may restrict a language $\mathcal{L}$ over Σ to all those words that only use letters from Σ' by forming

$$\{s \in \mathcal{L} \mid s \text{ only uses letters from } \Sigma'\}.$$

The definitions in Chapter 2 for forming new languages from existing ones are rigorous as given. You may want to improve your understanding of them by carrying out the following exercises.

Exercise A.10. (a) Calculate $\{a, b\}^2$ and $\{ab, ba\}^2$.

(b) Find the shortest set-theoretic expression you can find for the set of all words over the alphabet $\{0, 1, 2\}$ which consist of precisely three letters.

(c) Calculate $\{00, 11\}^*$.

(d) If $\Sigma_1 \subseteq \Sigma_2$ are two alphabets, argue that $\Sigma_1^* \subseteq \Sigma_2^*$.

A.2 Regular expressions

The formal definition of a pattern, and that of a word matching a pattern is given in Chapter 2. There are exercises in that chapter that encourage you to explore the recursive nature of the two definitions. The following should not be considered a formal exercise. Just think about it.

Exercise A.11. What would it take to demonstrate that a given word does *not* match a given pattern?

We have already seen that different patterns may describe the same language. In fact, we can come up with a number of rules of rewriting regular expressions in such a way that the language they define stays the same. Here are some examples, where p, p_1 and p_2 are arbitrary patterns over some alphabet Σ.

- $\mathcal{L}(\boldsymbol{\epsilon} p) = \mathcal{L}(p)$.
- $\mathcal{L}(p_1|p_2) = \mathcal{L}(p_2|p_1)$.
- $\mathcal{L}(p^*) = \mathcal{L}(\epsilon|pp^*)$.
- $\mathcal{L}((p_1|p_2)p) = \mathcal{L}(p_1p|p_2p)$.

Exercise A.12. Come up with three further such rules.

Exercise A.13. Argue that the equalities used in the recursive definition of a language matching a pattern on page 18 have to hold according to Definition 3.

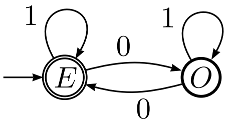
Exercise A.14. For Exercise 8 find set-theoretic notation for the sets you described in English.

Exercise A.15. Prove Proposition 2.1. *Hint: Take a closer look at the recursive definition of the language defined by a pattern above.*

A.3 Finite state automata

Matching up the formal definition of an automaton with its image is fairly straightforward.

As an example consider the picture on page 22



of an automaton that accepts precisely those words over $\{0, 1\}$ that contain an even number of 0s. We have two states, which in the picture are labelled E and O, so the set of all states is $\{E, O\}$. The initial state is E, and there is only one accepting state which is also E.

The transition function δ for this example is given by the following table:

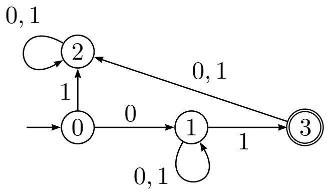
input	output
$(E, 1)$	E
$(E, 0)$	O
$(O, 1)$	O
$(O, 0)$	E

If we follow a word such as 001 through the automaton we can now do this easily: We start in state E and see a 0, so we next need to go to $\delta(E, 0)$ which is O. The next letter is 0 again, and from the new state O we need to go to $\delta(O, 0) = E$. The last letter is 1, so from the current state E we need to go to $\delta(E, 1) = E$. Since E is an accepting state we accept the word 001.

Exercise A.16. Describe the final picture from Section 3.1 in this way.

When moving to non-deterministic automata we no longer have a transition function, but a transition relation. Therefore we cannot use a table as above to write it out.

As an example, take the non-deterministic automaton from page 26.



It has states $\{0, 1, 2, 3\}$, start state 0 and accepting states $\{3\}$. The transition relation δ is defined as follows.

δ relates		to			
state	letter	0	1	2	3
0	0		✓		
0	1			✓	
1	0		✓		
1	1		✓		✓
2	0			✓	
2	1			✓	
3	0			✓	
3	1			✓	

Think of the ticks as confirming that from the given state in the left hand column there is an edge labelled with the letter in the second column to the state in the top row. We can see that δ here is a relation rather than a function because for the input state 1, letter 1, we find two ticks in the corresponding row.

Exercise A.17. Draw a non-deterministic automaton for the language described in Exercise 9 (d). Then describe it in the same way as the above example.

Exercise A.18. Draw the automata with states $\{0, 1, 2, 3\}$, start state 0, accepting states $\{0, 2\}$ and the following transitions.

(a)

δ relates		to			
state	letter	0	1	2	3
0	a		✓		
0	b			✓	
1	a				✓
1	b			✓	
2	a			✓	
2	b				✓
3	a				✓
3	b				✓

(b)

δ relates		to			
state	letter	0	1	2	3
0	a	✓	✓		
0	b	✓			
1	a	✓			✓
1	b	✓		✓	
2	a		✓		
2	b				
3	a	✓			
3	b		✓	✓	

Can you say anything about the resulting automata?

Exercise A.19. Using Algorithm 1 on page 33 we know how to turn a non-deterministic automaton into a deterministic one. Write out a definition of the transition function for the new automaton.

Exercise A.20. Prove Theorem 3.1. All that remains to be established is that a DFA generated by an NFA using Algorithm 1 from page 33 accepts precisely the same words as the original automaton.

Pictures are good and well when it comes to describing automata, but when such a description has to be given to a computer it is best to start from the mathematical description and turn it into a data structure.

Exercise A.21. What would it take to define an automaton class in Java? How would a class for deterministic automata differ from one for non-deterministic ones? *Hint: You may not know enough* Java *yet to decide this. If this is the case then return to the question later.*

Exercise A.22. Algorithm 2 is quite complicated. To try to understand it better I recommend the following for the general description which starts on page 41.

(a) Justify to yourself the expressions for $\mathcal{L}^{\leq j}_{j\to i}$ and $\mathcal{L}^{\leq i}_{j\to i}$ from the general ones for $\mathcal{L}^{\leq k}_{j\to i}$.

(b) Justify the equalities given at the end of the description of the general case.

Exercise A.23.* Explain why it is safe not to draw the 'dump' states for the two automata when constructing an automaton for the intersection of the languages recognized by the automata as described on page 54 by forming the product of the two automata.

Exercise A.24. Exercise 34 has two parts that are mathematical, namely a and (e). If you have delayed answering these do so now.

Let us now turn to the question of when to automata recognize the same language. We begin by giving a proper definition of what it means for two automata to be isomorphic.

Definition 21. *An automaton* $(Q, q_\bullet, F, \delta)$ *over the alphabet* Σ *is isomorphic to an automaton* $(P, p_\bullet, E, \gamma)$ *over the same alphabet if and only if there exists a function* $f\colon\ Q \longrightarrow P$ *such that*

- f *is bijective;*
- $f(q_\bullet) = p_\bullet$*;*
- q *is in* F *if and only if* $f(p)$ *is in* E *for all* $q \in Q$ *and all* $p \in P$*;*
- *in case the automata are*
  - deterministic*:* $f(\delta(q, x)) = \gamma(f(q, x))$ *for all* $q \in Q$ *and all* $x \in \Sigma$*;*
  - non-deterministic*:* $q \xrightarrow{x} q'$ *according to* δ *is true if and only if* $f(q) \xrightarrow{x} f(q')$ *according to* γ *also holds.*

Exercise A.25. Convince yourself that you understand this definition! Also convince yourself that two automata are isomorphic precisely when they have the 'same' picture (if the states remained unlabelled).

On page 56 we had a calculation used to turn the problem of equivalence of two automata into one of deciding whether two (different) automata recognize any words at all. You should be able to prove this, using definitions from the set-theory part of COMP111.

Exercise A.26. For subsets S and S' of some set T show that $S = S'$ if and only if $S \cap (T - S') = \emptyset$ and $S' \cap (T - S) = \emptyset$.

The notion of bisimulation deserves additional study and, indeed, this is a concept that is very important in concurrency. Here we restrict ourselves to one more (if abstract) example.

Exercise A.27. (a) Show that if two DFA accept the same language then there are simulations in both directions between them.

(b) Find two automata that accept the same language but which do *not* have simulations in both directions between them.

(c) Assume we are given an NFA, and the DFA that results from applying Algorithm 1 to that NFA. Show that there is always a simulation from the NFA to the corresponding DFA. *Hint: You need to do Exercise A.19 first. Use the 'element' relation between states.*

Exercise 38 is somewhat mathematical in the way it demands you to think about languages. Try to do as many of its parts as you can.

A.4 Grammars

Exercises in this section that asks you to reason mathematically are Exercises 44 and 45—they ask you to demonstrate something.

In Section 4.3 there is a method that takes an automaton and turns it into a context-free grammar. How can we see that the words generated by this grammar are precisely those accepted by the automaton?

If we have a word $x_1x_2\cdots x_n$ accepted by the automaton then we get the following for the grammar.

In the automaton:

We start in state $q_\bullet$ and go to

$$q_1 = \delta(q_\bullet, x_1).$$

Once we have reached state

$$q_i = \delta(q_{i-1}, x_i),$$

we go to

$$q_{i+1} = \delta(q_i, x_{i+1})$$

and keep going until we reach

$$q_n = \delta(q_{n-1}, x_n).$$

Now q_n is an accepting state.

For the grammar:

We have $S = q_\bullet \Rightarrow x_1q_1$.

Once we have reached the word

$$x_1x_2\cdots x_iq_i$$

we apply a production rule to obtain

$$x_1x_2\cdots x_ix_{i+1}q_{i+1},$$

where

$$q_{i+1} = \delta(q_i, x_{i+1}),$$

and keep going until we reach

$$x_1x_2\cdots x_nq_n.$$

Since we know that q_n is an accepting state we may now use the production rule $q_n \to \epsilon$ to obtain $x_1x_2\cdots x_n$.

Exercise A.28. Use a similar argument to reason that if s is a word over Σ generated by the grammar then it is also accepted by the automaton.

Exercise A.29* We have already demonstrated that every regular languages can be generated by a right-linear grammar. Try to show the opposite, that is, that every language generated by a right-linear grammar is regular.

COMP11212 Fundamentals of Computation
Exercise Sheet 1
For examples classes in Week 2

Exercises for this week

Exercises marked this week:
For Exercises 9, 10 and 13 begin by writing down some words that belong to the language, and then some that do not. Once you have constructed your solution check that it treats these words correctly. The demonstrators will ask to see this!

Exercise 4 (a), (c), (j), (k) on page 15
Exercise 9 (a)–(c) on page 19
Exercise 9 (d), (e)
Exercise 10 (a), (b) on page 20
Exercise 13 (a)–(c) on page 26

Do not forget that you need to show us **all your work**!

Exercises preparing the marked exercises
If you find the exercises we will mark difficult you should first try the easier exercises preparing them.

- Exercise 3 might help with **Exercise 4**.
- Exercise 11 and Exercise 12 should help with **Exercise 13**.
- Finding regular expressions and automata for a given language requires practice. There's no method for this!

Foundational exercises
The foundational exercises prepare important concepts appearing in Chapters 3 and 4.

Exercise 1
Exercise 2
Exercise 8 (a)–(c)

Optional exercises from Appendix A

The exercises in Sections A.1 and A.2.

Optional further suitable exercises

Exercise 5
Exercise 6
Exercise 7 (This exercise makes a connection between our regular expressions and those standard in Unix.)
Further parts of Exercises 4, 8, 9, 10, 13

COMP11212 Fundamentals of Computation
Exercise Sheet 2
For examples classes in Week 3

Exercises for this week

Exercises marked this week

For Exercises 13 and 14 begin by writing down some words that belong to the language, and then some that do not. Once you have constructed your solution check that it treats these words correctly. The demonstrators will ask to see this!

Exercise 13 (d), (e) on page 26
Exercise 14 (a), (b) on page 26
Exercise 16 on page 28
Exercise 21 (c) on page 34
Exercise 23 on page 35

Do not forget that you need to show us **all your work**!

Exercises preparing the marked exercises

If you find the exercises we will mark difficult you should first try the easier exercises preparing them.

- Exercise 11 and Exercise 12 should help with **Exercises 13** and **14**.
- Exercise 15 will help with understanding the difference between NFAs and DFAs, and so help with **Exercise 16**.
- **Exercise 21** is about applying Algorithm 1, which has an example spelt out in the notes. Simpler versions occur in Exercise 19 and parts (a) and (b) of the same exercise.
- **Exercise 23** mostly consists of tasks you should begin to find easy by now.

Foundational exercises

Exercise 17
Exercise 18
Exercise 20

Optional exercises from Appendix A

Exercises A.16–A.20

Optional further suitable exercises

Further parts of Exercises 14 and 21 as well as Exercise 22

COMP11212 Fundamentals of Computation
Exercise Sheet 3
For examples classes in Week 4

Exercises for this week

Exercises marked this week

You have to do at least two expansions (using the algorithm) in Exercise 25 before starting to read off languages. Note that there is another exercise asking you to apply Algorithm 2 on next week's sheet, so it pays to use this examples class to ensure that you understand it!

Exercise 25 (a) on page 43
Exercise 31 (a) on page 51
Exercise 36 (a) and (b) on page 55
Exercise 36 (c) and (d) *Note that part (d) requires having read the Appendix.*

The last exercise appears out of order from where it is in the notes, but all that is needed to solve it has been discussed in the lectures.
Do not forget that you need to show us **all your work**!

Exercises preparing the marked exercises

If you find the exercises we will mark difficult you should first try the easier exercises preparing them.

- Exercise 24 asks you to apply Algorithm 2 to simpler automata and should help with **Exercise 25**. There are also two worked examples in the notes.
- **Exercise 31** (a) asks you to apply Algorithm 3 which has a worked example in the notes.
- **Exercise 36** moves from two examples to the general case, but requires a little bit of lateral thinking.

Optional exercises from Appendix A

Exercises A.21–A.23

Optional further suitable exercises

Exercise 27
Exercise 28
Exercise 30
Further parts of Exercises 25, 26, 32 and 31, but some of those will be covered next week.

COMP11212 Fundamentals of Computation
Exercise Sheet 4
For examples classes in Week 5

Exercises for this week

Exercises marked this week:
You have to do at least three expansions (using the algorithm) in Exercise 26 before starting to read off languages.

Exercise 26 (b) on page 43
Exercise 32 (b) on page 51
Exercise 35 (a) on page 54
Exercise 37 (a) on page 60
Exercise 38 (a)–(c) on page 61

Do not forget that you need to show us **all your work**!

Exercises preparing the marked exercises
You might find the following helpful.

- There are lots of exercises involving Algorithm 2 to help with **Exercise 26**.
- The safe way of solving **Exercise 32** is to apply Algorithms 3, 4 and 1 in that order. There are plenty of examples and exercises in the notes.
- There is a worked example of the method used in **Exercise 35** in the notes.
- **Exercise 37** is preceded by an example.
- In order to answer **Exercise 38** you'll have to practice thinking in the right way. There's no fixed method for this.

Foundational exercises
Exercise 33

Optional exercises from Appendix A

Exercises A.24–A.27

Optional further suitable exercises

Exercise 34 as well as further parts of Exercises 32, 35 and 38

COMP11212 Fundamentals of Computation
Exercise Sheet 5
For examples classes in Week 6

Exercises for this week

Exercises marked this week:
Exercise 42 (c) on page 66
Exercise 43 (a)–(b) on page 67
Exercise 46 (a) on page 68
Exercise 47 (a) on page 72
Exercise 50 on page 75

Do not forget that you need to show us **all your work**!

Exercises preparing the marked exercises
You might want to note the following.

- Exercise 39 helps you understand how to derive a word from a grammar, and so to design grammars as in **Exercises 42** and **43**.
- **Exercise 42** (a)–(b) might be easier to start with than part (c).
- There is a worked example demonstrating what has to be done in **Exercise 46** in the notes.
- **Exercise 47** is preceded by an example in the notes. There is no method that will solve this problem generally; you will have to think about each one separately.
- **Exercise 50** is preceded by a general description of the method to be applied here.

Optional exercises from Appendix A

Exercises in Section A.4

Optional further suitable exercises

Exercise 40
Exercise 41
Exercise 44
Exercise 45
Exercise 48
Exercise 51
Exercise 49
Exercise 52
Further parts of Exercises 42, 43, 46 and 47