Formal Reasoning

Herman Geuvers

Partly based on the reader of fall of 2002 by Henk Barendregt and Bas Spitters,
and with thanks to Wim Gielen for his reader “Discrete Wiskunde.”
As of 2008, yearly revised and expanded by Engelbert Hubbers and Freek Wiedijk.
Translated fall of 2016 by Kelley van Evert.

August 26, 2025

Contents

Chapter 1

Propositional logic

In this chapter we discuss propositional logic, often also called propositional calculus.

1.1 Formal languages and natural languages

Natural languages, such as Dutch, English, German, etc., are not always as exact as one would hope. Take a look at the following examples:

- Socrates is a human being. Human beings are mortal. So, Socrates is mortal.
- I am someone. Someone painted the Mona Lisa. So, I painted the Mona Lisa.

The first statement is correct, but the second is not. Even though they share the same form.

And what about the sentence,

$\boxed{\text{This sentence is not true.}}$

Is it true, or not?

To avoid these kinds of problems, we use formal languages. Formal languages are basically laboratory-sized versions, or models, of natural languages. But we will see that these artificial languages, even though they are relatively simple, can be used to express statements and argumentations in a very exact and unambiguous manner. And this is very important for many applications, such as describing the semantics of a program. Such specifications should obviously not lead to misunderstandings.

To start, we will show how we make the transition from the English language to a formal language. In reasoning, we often combine small statements to form bigger ones, as in for instance: '*If* it rains *and* I'm outside, *then* I get wet.' In this example, the small statements are 'it rains', 'I'm outside', and 'I get wet.'

1.2 Dictionary

We can also formalize this situation with a small dictionary.

R	it rains
S	the sun shines
RB	there is a rainbow
W	I get wet
D	I stay dry
Out	I'm outside
In	I'm inside

So then, the sentence we introduced above becomes 'if R and Out, then W.' Similarly, we could form statements as: 'if RB, then S', 'S and RB', or 'if R and In, then D.'

1.3 Connectives

We can also translate the connectives, which we will do like this:

Formal language	English
$f \wedge g$	f and g
$f \vee g$	f or g, or both
$f \rightarrow g$	if f, then g
$f \leftrightarrow g$	f if and only if g[1]
$\neg f$	not f

Now, the sentences we formed above become 'RB → S', 'S ∧ RB', and '(R ∧ In) → D.'

English	Semi-formal	Formal
If it rains *and* I'm outside, *then* I get wet.	If R and Out, then W.	$(\mathrm{R} \wedge \mathrm{Out}) \rightarrow \mathrm{W}$
If there is a rainbow, *then* the sun shines.	If RB, then S.	$\mathrm{RB} \rightarrow \mathrm{S}$
I'm inside *or* outside, or both.	In or Out, or both.	$\mathrm{In} \vee \mathrm{Out}$

Exercise 1.A
Form sentences in our formal language that correspond to the following English sentences:

(i) It is neither raining, nor is the sun shining.
(ii) The sun shines unless it rains.
(iii) Either the sun shines, or it rains. (But not both simultaneously.)
(iv) There is only a rainbow if the sun is shining and it is raining.
(v) If I'm outside, I get wet, but only if it rains.

So far, we have modeled our symbols (like ∨ and ∧) to be very much like the English connectives. One of these English connectives is the word 'or,' which combines smaller statements into larger, new statements. However, the meaning of the English word 'or' can be a bit ambiguous. Take the following proposition: '$1 + 1 = 2$ or $2 + 3 = 5$.' Is it true, or not? It turns out that people sometimes differ in opinion on this, and when you think about it a bit, there are two distinct meanings that 'or' can have in the English language. Students of Information Science and Artificial Intelligence obviously don't like these ambiguities, so we will simply choose the meaning of ∨ to be one of the two usual meanings of 'or': we will agree that 'A or B' is also true in the case that both A and B are true themselves. Definition 1.6 will formalize this agreement.

Exercise 1.B
Can you also express $f \leftrightarrow g$ using the other connectives? If so, show how.

Exercise 1.C
Translate the following formal sentences into English:

(i) $\mathrm{R} \leftrightarrow \mathrm{S}$
(ii) $\mathrm{RB} \rightarrow (\mathrm{R} \wedge \mathrm{S})$
(iii) $\mathrm{Out} \rightarrow \neg\, \mathrm{In}$
(iv) $\mathrm{Out} \vee \mathrm{In}$

Definition 1.1
The language of propositional logic is defined to be as follows. Let A be an infinite collection of *atomic propositions* (also sometimes called *propositional variables* or *letters*):

$$A := \{a, b, c, d, a_1, a_2, a_3, \ldots\}$$

[1] Mathematicians will often simply write "iff" instead of the lengthier "if and only if."

Let V be the set of *connectives*:

$$V := \{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$$

Let H be the set of *parentheses*:

$$H := \{(,)\}$$

Then, let the *alphabet* be the set $\Sigma := A \cup V \cup H$, where '∪' stands for the union of two sets. If you are not familiar with set theory like this, please read Appendix A. For this chapter, it is probably not needed to study this, but for Chapters 2 and 4 it is essential that you understand these concepts.

Now, we can form the *words* of our language:

1. Any atomic proposition is a word.
2. If f and g are words, then so too are $(f \wedge g), (f \vee g), (f \rightarrow g), (f \leftrightarrow g)$, and $\neg f$.
3. All words are made in this way. (No additional words exist.)

We call the words of this language *propositions*.

Convention 1.2
Usually, we omit the outermost parentheses of a formula, thus for example writing $a \wedge b$ instead of $(a \wedge b)$. Of course we cannot always do the same with the inner parentheses, take for example the logically inequivalent $(\mathrm{R} \wedge \mathrm{S}) \rightarrow \mathrm{RB}$ and $\mathrm{R} \wedge (\mathrm{S} \rightarrow \mathrm{RB})$. What we can do, is *agree* upon a notation in which we are allowed to omit *some* of the parentheses. We do this by defining a *priority* for each connective:

- $\neg$ binds stronger than $\wedge$
- $\wedge$ binds stronger than $\vee$
- $\vee$ binds stronger than $\rightarrow$
- $\rightarrow$ binds stronger than $\leftrightarrow$

This means that we must interpret the formula $\mathrm{In} \vee \mathrm{RB} \rightarrow \mathrm{Out} \leftrightarrow \neg \mathrm{S} \wedge \mathrm{R}$ as the formula $((\mathrm{In} \vee \mathrm{RB}) \rightarrow \mathrm{Out}) \leftrightarrow (\neg \mathrm{S} \wedge \mathrm{R})$.

Only using these priorities is not enough though: it only describes where the implicit parentheses are in the case of *different* connectives. When statements are built up by repeated use of the *same* connective, it is not clear yet where these parentheses should be read.

For example, should we parse $\mathrm{Out} \rightarrow \mathrm{R} \rightarrow \mathrm{W}$ as expressing $(\mathrm{Out} \rightarrow \mathrm{R}) \rightarrow \mathrm{W}$ or as expressing $\mathrm{Out} \rightarrow (\mathrm{R} \rightarrow \mathrm{W})$? This is formalized by defining the *associativity* of the operators.

Convention 1.3
The connectives $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ are *right associative.* This means that, if $v \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$, then we must read

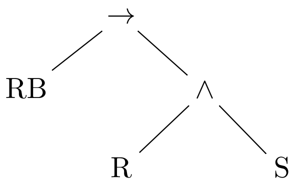
$$A \;\; v \;\; B \;\; v \;\; C$$

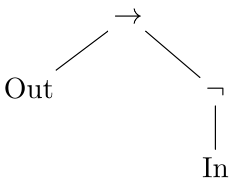
as expressing

$$A \;\; v \;\; (B \;\; v \;\; C)$$

Note, though, that this is a choice that is sometimes made differently, outside of this course. So sometimes, in other readers and books on logic, an other choice might be made.

Remark 1.4
We can also express the structure of the formula in *parse trees* where the atomic propositions are the leaves, and the logical operators the nodes. In addition, unary operators have a single arrow downwards and binary operators have two arrows downwards, reflecting the left and the right operand of the operator. The formulas $\mathrm{RB} \to (\mathrm{R} \wedge \mathrm{S})$ and $\mathrm{Out} \to \neg\, \mathrm{In}$ are represented respectively as:





Note that parentheses are not shown in the tree, because the structure is already provided by the tree itself. And also note that the parentheses in $\mathrm{RB} \to (\mathrm{R} \wedge \mathrm{S})$ were not needed by Convention 1.2 anyway!

Remark 1.5
(This remark may only have meaning to you after reading Chapter 4, "Languages and automata"; so after reading that chapter, you might want to reread this remark.) The formal definition of our language, with the help of a *context-free grammar*, is as follows: Let $\Sigma = A \cup V \cup H$, that is to say $\Sigma = \{a, b, c, \dots, \neg, \wedge, \vee, \to, \leftrightarrow, \neg, (,)\}$. Then the language is defined by the grammar

$$S \ \to \ a \mid b \mid c \mid \dots \mid \neg S \mid (S \wedge S) \mid (S \vee S) \mid (S \to S) \mid (S \leftrightarrow S)$$

Obviously, in stead of the dots all other symbols of the alphabet A should be listed.

1.4 Meaning and truth tables

The sentence 'if a and b, then a' is true, whatever you substitute for a and b. So, we'd like to be able to say: the sentence '$a \wedge b \to a$' is true[2]. But we can't, because we haven't formally defined what that means yet. As of yet, '$a \wedge b \to a$' is only one of the words of our formal language. Which is why we will now turn to defining the meaning of a word (or statement) of language, specifically speaking, when such a statement of logic would be *true*.

For the atomic propositions, we can think of any number of simple statements, such as '2 = 3,' or 'Jolly Jumper is a horse,' or 'it rains.' In classical logic, which is our focus in this course, we simply assume that these atomic propositions may be true, or false. We don't further concern ourselves with the specific truth or falsity of particular statements like 'on January the 1st of 2050 it will rain in Nijmegen.'

The truth of atomic propositions will be defined solely by their interpretation in a *model*. For example, '2 = 3' is not true in the model of natural numbers. And 'Jolly Jumper is a horse' is true in the model that is a comic of Lucky Luke. And the sentence 'it rains' was not true in Nijmegen, on the 17th of September of 2002.

Now let's take a look at the composite sentence '$a \wedge b$.' We would want this sentence to be true in a model, exactly in the case that both a and b are true in that model. If we then just enumerate the possible truth values of a and of b, we can define the truth of $a \wedge b$ in terms of these.

In Computing Science, we often simply write 1 for *true*, and 0 for *false*. Logical operations, then, are the elementary operations on bits.

So, we get the following truth table:

[2] Note: $a \wedge b \to a$ should be read as $(a \wedge b) \to a$

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

And what of the ‘if. . . , then. . . ’ construction to combine statements into larger ones? Here, too, natural language is somewhat ambiguous at times. For example, what if A is false, but B is true. Is the sentence ‘if A, then B’ true? Examples are:

‘if $1 + 1 = 3$, then $2 + 2 = 6$’
‘if I jump off the Erasmus building, I’ll turn into a bird’
‘if I understand this stuff, my name is Alpje’

We will put an end to all this vagueness by simply *agreeing* upon the truth tables for the connectives (and writing ‘$A \to B$’ instead of ‘if A, then B’), in this next definition.

Definition 1.6
The *truth tables* for the logical connectives are defined to be:

x	$\neg x$
0	1
1	0

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \to y$
0	0	1
0	1	1
1	0	0
1	1	1

x	y	$x \leftrightarrow y$
0	0	1
0	1	0
1	0	0
1	1	1

Using our truth tables, we can determine the truth value of complex propositions from the truth values of the individual atomic propositions. We do this by writing out larger truth tables for these complex propositions.

Example 1.7
This is the truth table for the formula $a \vee b \to a$:

a	b	$a \vee b$	a	$a \vee b \to a$
0	0	0	0	1
0	1	1	0	0
1	0	1	1	1
1	1	1	1	1

Exercise 1.D
Draw the parse trees and give the truth tables for:

(i) $a \vee \neg a$
(ii) $(a \to b) \to a$
(iii) $a \to (b \to a)$
(iv) $a \wedge b \to a$
(v) $a \wedge (b \to a)$
(vi) $\neg a \to \neg b$

1.5 Models and truth

In the introduction, we spoke of *models* and truth in models. In the truth tables, we have seen that the '*truth*' of a proposition is fully determined by the values that we assign to the atomic propositions. A model of propositional logic is therefore simply some assignment of values ($\{0, 1\}$) to the atomic propositions.

Definition 1.8
A *model* of propositional logic is an *assignment*, or *valuation*, of the atomic propositions: a function $v : A \to \{0, 1\}$.

To determine the truth value of a proposition f, we don't really need to know the value of all atomic propositions, but only of those of which f is comprised. Therefore, we will actually just equate a model to a *finite assignment* of values.

Example 1.9
In the model v that has $v(a) = 0$ and $v(b) = 1$, the proposition $a \vee b \to a$ has the value 0.

Convention 1.10
If v is a model, we will also simply write $v(f)$ for the value that f is determined to have under that model. We say that f is *true in a model* v in the case that $v(f) = 1$. We also use the phrase v *is a model of* f to indicate that $v(f) = 1$.

So $a \vee b \to a$ is not true, or false, in v when $v(a) = 0$ and $v(b) = 1$. However, $a \to (b \to a)$ is true in such a model.

Definition 1.11
If a proposition f is true in every conceivable model (which is to say that in its truth table, there are only 1's in its column), then we call that proposition *logically true*, *logically valid* or just *valid*. The notation for this is: $\models f$. A logically true statement is also called a *tautology*.

If a proposition f is *not* logically true, that can be denoted by writing $\not\models f$.

Exercise 1.E
Which of the following propositions are logically true?

(i) $a \vee \neg a$
(ii) $a \to (a \to a)$
(iii) $a \to a$
(iv) $(a \to b) \to a$
(v) $a \to (b \to a)$
(vi) $a \wedge b \to a$
(vii) $a \vee (b \to a)$
(viii) $a \vee b \to a$

Exercise 1.F
Let f and g be *arbitrary* propositions. Find out whether the following statements hold. Explain your answers.
(i) If $\models f$ and $\models g$, then $\models f \wedge g$.
(ii) If not $\models f$, then $\models \neg f$.
(iii) If $\models f$ or $\models g$, then $\models f \vee g$.
(iv) If (if $\models f$, then $\models g$), then $\models f \to g$.
(v) If $\models \neg f$, then not $\models f$.
(vi) If $\models f \vee g$, then $\models f$ or $\models g$.
(vii) If $\models f \to g$, then (if $\models f$, then $\models g$).
(viii) If $\models f \leftrightarrow g$, then ($\models f$ if and only if $\models g$).
(ix) If ($\models f$ if and only if $\models g$), then $\models f \leftrightarrow g$.

Remark 1.12
Basically, we can distinguish three types of truth:

1. A formula f is true in a given model, so $v(f) = 1$. Typically, this is related to the idea of 'formula f is true, right here, right now'.
2. A formula f is true given a specific dictionary for the symbols. For instance, the formula In $\leftrightarrow \neg$ Out is true in the dictionary in Section 1.2 as (in general) 'being inside' has the same meaning as 'not being outside'. However, if we would use a different dictionary like

Out	I have a blackout
In	I'm inside

   then the formula In $\leftrightarrow \neg$ Out is no longer true, as 'being inside' has nothing to do with 'not having a blackout'.
3. A formula f is true in *all* models. for instance, the formula In $\rightarrow \neg\neg$ In is true in all models, independent of the dictionary being used. In particular, this is the *logically true* from Definition 1.11.

The first and the third are technical things in logic. The second does not mean anything in logic. For instance, what if someone is in the doorway? Is this person inside, outside, both, or neither?

1.6 Logical equivalence

Definition 1.13
Two propositions f and g are said to be *logically equivalent* in the case that f is true in a model if and only if g is true in that same model.

To formulate this more precisely: f and g are logically equivalent if for every model v it holds that $v(f) = 1$ if and only if $v(g) = 1$. This boils down to saying that f and g have the same truth tables.

To denote that f and g are logically equivalent, we write $f \equiv g$.

Often, a proposition can be replaced by a simpler, equivalent proposition. For example, $a \wedge a$ is logically equivalent to a. So, $a \wedge a \equiv a$.

Exercise 1.G
For each of the following couples of propositions, show that they are logically equivalent to each other.

(i) $(a \wedge b) \wedge c$ and $a \wedge (b \wedge c)$

(ii) $(a \vee b) \vee c$ and $a \vee (b \vee c)$

In this case, apparently, the placement of parentheses doesn't really matter. For this reason, in some cases we simply omit such superfluous parentheses. In Convention 1.3 we agreed that all binary connectives would be right associative. But here, we see that this choice, at least for $\wedge$ and $\vee$, is arbitrary: had we agreed that $\wedge$ and $\vee$ associate to the left, then that would have had no consequence for the truth value of the composite propositions.[3] In Exercise 1.D we have seen that for the connective $\rightarrow$, it actually does matter in which direction it associates!

[3] The reason that we chose for the right associativity of $\wedge$ and $\vee$ is to be consistent with the proof system Coq that will be used in the course "Logic and Applications".

Remark 1.14
The propositions $a \wedge b$ and $b \wedge a$ are mathematically speaking logically equivalent. In English though, the sentence 'they married and had a baby' often means something entirely different compared to 'they had a baby and married.'

Here are a number of logical equivalences that demonstrate the distributivity of the operators $\neg$, $\wedge$, and $\vee$ over parentheses. These equivalences are called *logical laws*.

$\neg(f \wedge g) \equiv \neg f \vee \neg g$ $\neg(f \vee g) \equiv \neg f \wedge \neg g$	Laws of *De Morgan*
$f \wedge (g \vee h) \equiv f \wedge g \vee f \wedge h^4$ $f \vee g \wedge h \equiv (f \vee g) \wedge (f \vee h)$	Laws of distributivity
$\neg\neg f \equiv f$	Double negation elimination (DNE)
$f \to g \equiv \neg g \to \neg f$	Law of contraposition
$f \to g \equiv \neg f \vee g$	Material implication

Exercise 1.H
Let f and g be propositions. Is the following statement true? $f \equiv g$ if and only if $\models f \leftrightarrow g$.

1.7 Logical consequence

In English, the statement that 'the sun is shining' follows logically from the statement that 'it is raining and the sun is shining.' Now, we want to define this same logical consequence for our formal language: that a is a logical consequence of $a \wedge b$.

Definition 1.15
A proposition g is a *logical consequence*, also sometimes called a *logical entailment*, of the proposition f, if g is true in every model for which f is true. Said differently: a proposition g is a logical consequence of f if, in every place in the truth table of f in which there is a 1, the truth table of g also has a 1. Notation: $f \models g$. Yet another way of phrasing this is stating that g is true in all models of f.

If g is *not* a logical consequence of f, that can be denoted by $f \not\models g$.

Exercise 1.I
Are the following statements true?

(i) $a \wedge b \models a$

(ii) $a \vee b \models a$

(iii) $a \models a \vee b$

(iv) $a \wedge \neg a \models b$

Theorem 1.16
Let f and g be propositions. Then the following holds:

$$\models f \to g \textit{ if and only if } f \models g.$$

You should now be able to find a proof for this proposition. [*Hint:* Take a look at Exercises 1.F and 1.I.]

Remark 1.17
Note that the symbols $\models$ and $\equiv$ were not included in the definition of the propositions of our formal language. These symbols are not a part of the language, but are merely used to speak mathematically *about* the language. Specifically, this means that constructions as '$(f \equiv g) \to (\models g)$' and '$\neg \models f$' are simply meaningless, and we should try to avoid writing down such invalid constructions.

Remark 1.18
For more on logic, you can have a look at books like [4] and [2].

1.8 Important concepts

Chapter 2

Predicate logic

"A woman is happy if she loves someone,
and a man is happy if he is loved by someone."

Instead of discussing the truth of this sentence, we are going to find out how to write it in a formal way. Hopefully, this will then help us analyze the truth of such sentences. To jump right in, we will give a formal account of the sentence after defining the dictionary:

W	set of (all) women
M	set of (all) men
$L(x, y)$	x loves y
$H(x)$	x is happy

Now the formal translation becomes:

$$\begin{gathered} \forall w \in W \Big[\big(\exists x{\in}(M \cup W)\ L(w, x)\big) \to H(w) \Big] \\ \wedge \\ \forall m \in M \Big[\big(\exists x{\in}(M \cup W)\ L(x, m)\big) \to H(m) \Big] \end{gathered} \tag{2.1}$$

The '$\forall$' symbol stands for "for all", and the '$\exists$' symbol stands for "there exists". The '$\in$' symbol denotes the membership of an element in a set, and '$\cup$', which we have encountered before, stands for the union[1] of two sets. If we translate the formal sentence back to English, in a very literal way, we get:

> For every woman it holds that, if there is a person that she loves, she is happy, and
> for every man it holds that, if there is person that loves him, he is happy.

You should now check whether this is indeed the same as the sentence that we started with, and whether you can see how this is indeed represented in the formal translation above. Note our use of the square parentheses [_] (also called brackets.) This is only for readability though, so that you can easily see which parenthesis belongs to which.
Because it enables us to translate a sentence, we will also call our dictionary an *interpretation.*

[1] Note that there are three main binary operators on sets: $A \cup B$, the *union* of A and B is the set where each element is in A, B, or in both; $A \cap B$, the *intersection* of A and B, is the set where each element is both in A and B; $A \setminus B$, the *set difference* of A and B, is the set where each element is in A but not in B. Note that whereas $A \cup B = B \cup A$ and $A \cap B = B \cap A$, in general $A \setminus B \neq B \setminus A$. For instance, let $A = \{0, 2, 4, 6\}$ and let $B = \{0, 3, 6\}$, then $A \cup B = \{0, 2, 3, 4, 6\}$, $A \cap B = \{0, 6\}$, $A \setminus B = \{2, 4\}$, and $B \setminus A = \{3\}$. See Appendix A for more background information about sets.

2.1 Predicates, relations, and constants

Previously, we saw how propositional logic allowed us to translate the English sentence

If Sharon is happy, Koos is not.

as

$$SH \to \neg KH$$

by choosing the dictionary:

SH	Sharon is happy
KH	Koos is happy

But suppose we add more people to our statements, like Joris, and Maud. We will easily end up with an inconveniently lengthy dictionary soon, because we have to add an atomic proposition for every new person (JH, MH, …).

This is why we now take a better look at the form of the statement that

Sharon is happy

and find out that it is of the shape of a *predicate* $H(\ )$ applied to a *subject* s (Sharon). Let's write that as $H(s)$. The immediate benefit is that we can now also write the very similar

$$H(k),\ H(j), \text{ and } H(m)$$

for the subjects k, j, and m. Instead of subjects, we will often speak of *constants*.

Also for our list of subjects, we will use our dictionary to formally denote which subject belongs to which name. Moreover, we will indicate to which *domain* each of our subjects belongs.

s	Sharon ∈ "women"
k	Koos ∈ "men"
j	Joris ∈ "men"
m	Maud ∈ "women"

Maybe the real benefit of our new system is not yet entirely clear. (Now that we have one predicate H added to four subjects s, k, j, m, making a total of five symbols, which is more than the four we started out with: SH, KH, JH, MH…) But suppose now that our subjects would gain a number of other qualities, such as:

$T(x)$	x is tall
$B(x)$	x is beautiful
$N(x)$	x is nice
$I(x)$	x is intelligent

Moreover, besides our predicates (qualities), we also have (binary) relations, like the one we used above:

$$L(x, y) : \ x \text{ loves } y.$$

Now we can take the following sentence

Sharon is an intelligent beautiful woman;
and there is a nice tall guy who loves this character.

and formalize it with the formula

$$\begin{aligned} I(s) \quad \wedge \quad & B(s) \\ \wedge \quad & \exists x \in M \left[N(x) \wedge T(x) \wedge \exists y \in W \; [L(x,y) \wedge B(y) \wedge I(y)] \right]. \end{aligned}$$

... or did we mean to say,

$$\begin{aligned} I(s) \quad \wedge \quad & B(s) \\ \wedge \quad & \exists x \in M \; [N(x) \wedge T(x) \wedge L(x,s)]? \end{aligned}$$

Notice that we take these two statements in the sentence above, and join them together to a single formula with the '$\wedge$' symbol. And that, although the formula doesn't state that Sharon is a woman, that it doesn't need to, because we already defined that $s \in$ "women" in the dictionary. (So we don't need to add something like for example "$s \in W$" in the formula.)

The difference between the first and the second formal translation lies in what what referred to as "this character." (This might refer to "Sharon," or it might refer to "intelligent and beautiful woman.") It is not logic that decides such questions, but logic does make it explicit that this choice has to be made. Said differently: it makes explicit the ambiguities that often occur in the English (or any other natural) language.

Exercise 2.A
Give two possible translations for the following sentence.

Sharon loves Maud; a nice man loves this intelligent character.

Now let's translate the other way around, and decode a formula.

Exercise 2.B
Translate the following sentences to English.

(i) $\exists x \in M \left[T(x) \wedge \exists w \in W \; [B(w) \wedge I(w) \wedge L(x,w)] \right]$

(ii) $\exists x \in M \left[T(x) \wedge \exists w \in W \; [B(w) \wedge \neg I(w) \wedge L(x,w)] \wedge \exists w' \in W [I(w') \wedge L(w',x)] \right]$

2.2 The language of predicate logic

We will now define formally what the language of predicate logic looks like.

Definition 2.1
The language of predicate logic is built up from the following ingredients:

1. *Variables*, usually written $x, y, z, x_0, x_1, \ldots$, or sometimes $v, w, \ldots$
2. *Individual constants*, also called 'names', usually $a, b, c, \ldots$,
3. *Domains*, such as $M, W, E, \ldots$,
4. *Relation symbols*, each with a fixed "arity", that we sometimes explicitly annotate them with, as in P^1, R^2, T^3, ... So this means that P^1 takes a single argument, R^2 takes two arguments, etc.
5. The *atoms*, or *atomic formulas*, are $P^1(t)$, $R^2(t_1, t_2)$, $T^3(t_1, t_2, t_3)$ etc., in which t, t_1, t_2, t_3, etc. are either variables or individual constants.
6. The *formulas* are built up as follows:

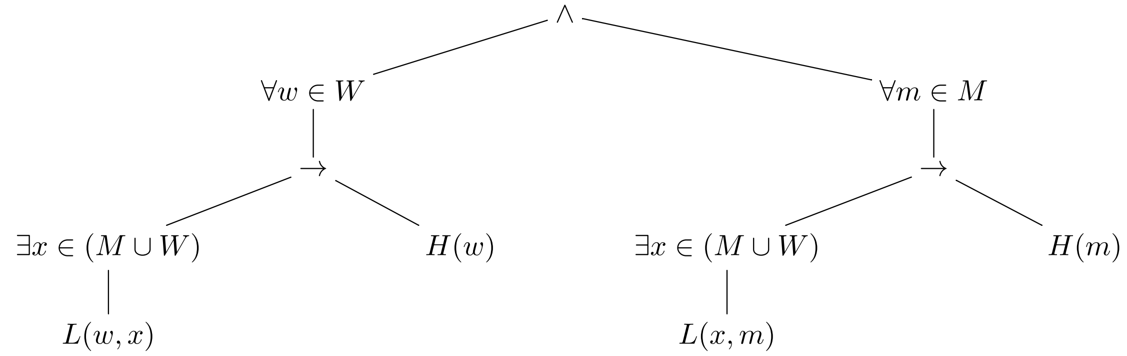
- Every atomic formula is a formula.
- If f and g are formulas, then so are $(f \wedge g)$, $(f \vee g)$, $(f \to g)$, $(f \leftrightarrow g)$, and $\neg f$.
- If f is a formula, and x is a variable, and D is a domain, then $(\forall x \in D\ f)$ and $(\exists x \in D\ f)$ are also formulas, made with the *quantifiers* '$\forall$' and '$\exists$'.
- All formulas are made in this way. (No others exist.)

Convention 2.2
Just as in propositional logic, we usually omit the outermost parentheses. And to be able to omit excessive parentheses within formulas, we expand upon our previous Convention 1.2 for propositional logic, adding the following:

- $\forall$ and $\exists$ bind stronger than all other connectives.

Somewhat opposing what we said earlier, in the case of $\forall$ and $\exists$, we add brackets for readability. So instead of $(\forall x \in D\ f)$, as in the definition, we write $\forall x \in D\ [f]$. And if we have a consecutive series of the same quantifiers over the same domain, we may group these quantifications: $\forall x \in D\ [\forall y \in D\ [f]]$ may be abbreviated to $\forall x, y \in D\ [f]$. Note that you are not allowed to combine existential and universal quantifications in such an abbreviation.

Remark 2.3
Just like we did in propositional logic, we can also express the structure of the formulas of predicate logic as parse trees, where the atoms are the leaves, and the logical operators the nodes. Formula (2.1) is represented by this tree:



And again, all parentheses and/or brackets that were used to show the structure of the formula are removed, because this structure is already provided by the tree itself. Obviously, parentheses that are used to indicate the arguments of relation symbols like H and L should not be removed.

Note that, if you look back at formula (2.1) in this chapter, you can see that it actually quite inconsistently uses parentheses and brackets. If we write it out as per the official definition, it would read:

$$\Big(\Big(\forall w \in W\ \big((\exists x{\in}(M \cup W)\ L(w,x)) \to H(w)\big)\Big) \wedge \Big(\forall m \in M\ \big((\exists x{\in}(M \cup W)\ L(x,m)) \to H(m)\big)\Big)\Big)$$

You can see why we prefer to somewhat liberally use parentheses and brackets for readability. Let us take a look at the meaning of all the parentheses in this official version of the formula:

$$\underbrace{\left(\underbrace{\left(\forall w \in W \underbrace{\left(\underbrace{\left(\exists x{\in} \underbrace{(M \cup W)}_{1}\ L \underbrace{(w,x)}_{2} \right)}_{3} \to H \underbrace{(w)}_{2} \right)}_{4} \right)}_{5} \land \underbrace{\left(\forall m \in M \underbrace{\left(\underbrace{\left(\exists x{\in} \underbrace{(M \cup W)}_{1}\ L \underbrace{(x,m)}_{2} \right)}_{3} \to H \underbrace{(m)}_{2} \right)}_{4} \right)}_{5} \right)}_{6}$$

1. These parentheses are used for the readability of the domain, which is the union of the sets M and W in this case. (The parentheses are not required by Definition 2.1 and hence they could have been left out in the parse tree in Remark 2.3 as well.)
2. These parentheses are needed because $L(w, x)$, $H(w)$, $L(x, m)$, and $H(m)$ are atomic formulas, so the parentheses are required.
3. These parentheses are needed because the $\exists$ quantifier needs them.
4. These parentheses are needed because the $\to$ requires them.
5. These parentheses are needed because the $\forall$ quantifier needs them.
6. These parentheses are needed because the conjunction with $\land$ requires them.

If we use square brackets as noted above, consistently writing $\forall x \in D\ [f]$ instead of $(\forall x \in D\ f)$, and furthermore omit all unnecessary parentheses according to our convention, we end up with the formula:

$$\forall w \in W \left[\exists x{\in}(M \cup W)\,[L(w,x)] \to H(w) \right] \land \forall m \in M \left[\exists x{\in}(M \cup W)\,[L(x,m)] \to H(m) \right].$$

Remark 2.4
Grammar of predicate logic.

Just as with propositional logic we can define the language of predicate logic as a formal language, with a grammar. This is done as follows. (You should reread this after having met grammars in Chapter 4 about languages.)

Individual := Variable | Name
Variable := $x, y, z, x_1, y_1, z_1, \ldots$
Name := $a, b, c, d, e, \ldots$
Domain := $D, E, \ldots$
Atom := P^1(Individual) | R^2(Individual,Individual)
| T^3(Individual,Individual,Individual) | ...
Formula := Atom
| ¬ Formula
| (Formula → Formula) | (Formula ∧ Formula)
| (Formula ∨ Formula) | (Formula ↔ Formula)
| (∀ Variable ∈ Domain Formula)
| (∃ Variable ∈ Domain Formula)

Exercise 2.C
Formalize the following sentence.

Sharon is beautiful; there is a guy who feels good about himself whom she loves.

Here, we will treat feeling good about oneself as being in love with oneself.

Exercise 2.D
Formalize the following sentences:

(i) *For every two persons we have: the first one loves the second one only if the first one feels good about him- or herself.*

(ii) *For every two persons we have: the first one loves the second one if this second person feels good about him- or herself.*

(iii) *For every two persons we have: the first one loves the second one exactly in the case that the second one feels good about him- or herself.*

(iv) *There is somebody who loves everyone.*

2.3 Truth determination

Take a look at the following two formulas

$$\begin{aligned} F_1 &= \forall x{\in}D \exists y{\in}D \; [K(x,y)] \\ F_2 &= \exists x{\in}D \forall y{\in}D \; [K(x,y)] \end{aligned}$$

(Note how we cut back on parentheses.) When is a formula *true*? Truth is relative and depends upon an *interpretation* and a *structure* .

Definition 2.5
A *structure* is that piece of the 'real' world in which formulas gain meaning by means of an *interpretation*.[2] A *model* is a pair (M, I) where M is a structure and I an interpretation.

Because we haven't yet explained what an interpretation is, this definition isn't really conclusive. But before we introduce the concept of an interpretation, let us first explain with a few examples what we mean with 'a piece of the real world.' Central here is the choice of the domains that we choose, or how we restrict ourselves to a certain part of them, and which predicates, relations, and constants are known within those domains.

1. Structure M_1

Domain(s)	all students in the lecture hall
Predicate(s)	is female
	is more than 20 years old
Relation(s)	has a student number lower than
	is not older than
	is sitting next to
Constant(s)	...

2. Structure M_2

[2] Note that in previous editions of this course a 'structure' was called a 'model' and what is called a 'model' this year, had no specific name in previous years. And 'interpretations' are still the same things as before.

Domain(s)	people
	animals
Predicate(s)	is female
	is human
	is canine
Relation(s)	owns
	is older than
Constant(s)	Maud
	Sharon
	Joris
	Koos
	Sif
	Beatrice

In the case of the students in the lecture hall, we can determine which are female and not. And if Sharon and Maud would happen to be present, which we could indicate by including their names as constants in the structure, it could be determined whether they are sitting next to each other. In the other case, there are biologists who can determine for any animal (like the cats Sif and Beatrice) whether it is a dog or not, and furthermore it is usually easy to establish whether someone is the owner of a certain animal. In which way this is done, should actually still be formally agreed upon, if we wish to have an exact structure.

The important aspect is that, if we wish to use a certain part of the world as a domain for predicate logic, we should have a conclusive and consistent way of determining the truth of statements about predicates over subjects and relations between subjects of the domain.

Now we can define the notion of an interpretation.

Definition 2.6
An *interpretation* is given by a dictionary, which states:

1. Which sets are referred to by the domain symbols,
2. which subjects are referred to by the names (and to which domain sets these belong),
3. and which predicates and relations are referred to by the predicate and relation symbols.

In particular, then, the interpretation establishes a clear connection between the symbols in formulas and the structure that we are looking at. In the literature, the interpretation is therefore often done via an interpretation function. See for example [4].

Definition 2.7
A formula f is called true in a structure under a given interpretation if the translation given by the interpretation is actually true in the structure.

Example 2.8
In the running example of Sharon, Koos, Joris, and Maud, the interpretation has been given in the dictionary we introduced: we know exactly what is meant with the formulas $L(s, k)$ and $\exists x \in M\ L(x, s)$. The structure is the factual situation involving these people, in which we can indeed determine whether $L(s, k)$ happens to be true (whether Sharon loves Koos), and whether $\exists x \in M\ L(x, s)$ happens to be true (whether there is some guy who loves Sharon).

Whether the formulas F_1 and F_2 are true within M_1, can thus only be determined if we also define an interpretation which gives meaning to the symbols. Here are three different possible interpretations:

1. Interpretation I_1

D	all students in the lecture hall
$K(x,y)$	x has a student number that is lower than that of y

2. Interpretation I_2

D	all students in the lecture hall
$K(x,y)$	x is not older than y

3. Interpretation I_3

D	all students in the lecture hall
$K(x,y)$	x is sitting next to y

We can assess the truth of the formulas F_1 resp. F_2, in these structures and under the given interpretations, by checking whether the propositions hold that these formulas express, within the structures, and under the interpretations.

Exercise 2.E

(i) Verify that F_2 does not hold in M_1 under the interpretation I_1. But does F_1 hold?

(ii) Verify that F_1 holds in M_1 under the interpretation I_2. Does F_2 hold as well?

(iii) Check whether F_1 in M_1 is true under the interpretation of I_3, by looking around in class. And check whether F_2 is true or not, as well.

Because structure M_1 only mentions a single domain, it would seem as if every interpretation would have to speak of the same domain. But this is not the case, as we can see in the following interpretations for structure M_2:

1. Interpretation I_4

D	people
$K(x,y)$	x isn't older than y

2. Interpretation I_5

D	people as well as animals
$K(x,y)$	x owns y

Note that in the case of I_5, that if x is an animal and y is a person, the statement $K(x,y)$ is automatically false, because animals don't own people.

3. Interpretation I_6

D	animals
$K(x,y)$	y is older than x

Let's now take a look at the formulas G_1 and G_2, and two structures in which we will interpret them. (Note their difference to F_1 and F_2.)

$$\begin{aligned} G_1 &= \forall x{\in}D \exists y{\in}D\ [K(x,y)] \\ G_2 &= \forall x{\in}D \exists y{\in}D\ [K(y,x)] \end{aligned}$$

Example 2.9

We define the structures M_3

Domain(s)	Natural numbers, $\mathbb{N} = \{0,1,2,3,\ldots\}$
Relation(s)	smaller than ($<$)

and M_4

Domain(s)	Rational numbers (fractions), $\mathbb{Q}$, (for example $-\frac{1}{2}$, $3(=\frac{3}{1})$, $0,\ldots$)
Relation(s)	smaller than ($<$)

And two self-evident accompanying interpretations, namely interpretation I_7

D	$\mathbb{N}$
$K(x,y)$	$x<y$

and interpretation I_8

D	$\mathbb{Q}$
$K(x,y)$	$x<y$

In structure M_3 under interpretation I_7 the formula G_1 is true. Indeed, for every number $x \in \mathbb{N}$ we can easily find a number $y \in \mathbb{N}$, for example $y = x+1$, such that $x < y$. In structure M_4 under interpretation I_8 the formula G_1 is true as well. Again, we can take $y = x+1$ and $x < y$ holds. Whenever we have such a method in which we can state precisely how to obtain a y for any x, we speak of having an *algorithm* or *recipe*.

Convention 2.10
If a formula f is true in a structure M under the interpretation I, we denote this by

$$(M, I) \models f$$

Using the pair notation, we have seen that:

$$\begin{aligned}(M_3, I_7) &\models G_1\\ (M_4, I_8) &\models G_1\end{aligned}$$

If we hadn't given the structures M_3 and M_4 a name, we could also have simply written this as follows:

$$\begin{aligned}((\mathbb{N},<), I_7) &\models G_1\\ ((\mathbb{Q},<), I_8) &\models G_1\end{aligned}$$

Here, the tuple $(\ldots)$ is a shorthand notation simply listing the domains, the predicates, the relations, and the constants without explicitly stating what is what. It is typically only used for structures with numerical domains.

Because the (important part of the) structure is often already expressed in the given interpretation, we will often omit an exact definition of the structure.

Exercise 2.F
Verify that G_2 is indeed true in structure M_4 under the interpretation I_8, but not in structure M_3 under the interpretation I_7. Stated differently: verify that $((\mathbb{Q},<), I_8) \models G_2$ and verify that $((\mathbb{N},<), I_7) \not\models G_2$.

Exercise 2.G
Define the interpretation I_9 as:

D	$\mathbb{N}$
$K(x,y)$	$x = 2 \cdot y$

Are the formulas G_1 and/or G_2 true under this interpretation?

Exercise 2.H
Define the interpretation I_{10} as:

D	$\mathbb{Q}$
$K(x,y)$	$x = 2 \cdot y$

Are the formulas G_1 and/or G_2 true under this interpretation?

Exercise 2.I
We take as structure the countries of Europe, and the following interpretation I_{11}:

E	the set of countries of Europe
n	The Netherlands
g	Germany
i	Ireland
$B(x, y)$	x borders y
$T(x, y, z)$	x, y, and z share a tripoint (where the borders of all three countries meet)

(i) Formalize the sentence "The Netherlands and Germany share a tripoint."
(ii) Which of the following formulas are true in this structure and under this interpretation?
  (1) $G_3 := \forall x \in E\ \exists y \in E\ [B(x, y)]$
  (2) $G_4 := \forall x, y \in E\ [(\exists z \in E\ T(x, y, z)) \to B(x, y)]$
  (3) $G_5 := \forall x \in E\ [B(i, x) \to \exists y \in E\ [T(i, x, y)]]$.

Exercise 2.J
Find a structure M_5 and an interpretation I_{12} such that this formula holds:

$$(M_5, I_{12}) \models \forall x \in D\ \exists y \in E\ [R(x, y) \wedge \neg R(y, x) \wedge \neg R(y, y)]$$

In Convention 2.10 we have seen that we write $(M, I) \models f$ in the case that the formula f of predicate logic (with or without equality, which will be discussed in Section 2.4) is true in a structure M under the translation given by an interpretation (or, dictionary) I. Now, we will expand a bit upon this definition.

Definition 2.11
A formula f of predicate logic f is said to be *logically true*, or *logically valid*, which we then denote $\models f$, when for *any* structure and *any* interpretation, the translation holds in that structure.

We often omit the *logically* part, simply writing that f is *true* or f is *valid* instead of writing that f is *logically true* or *logically valid*.

Example 2.12
Consider the following statements:
(i) $\models \forall x{\in}D\ [P(x) \to P(x)]$.
(ii) $\models (\exists x{\in}D \forall y{\in}D\ [P(x, y)]) \to (\forall y{\in}D \exists x{\in}D\ [P(x, y)])$.
(iii) $\not\models (\forall y{\in}D \exists x{\in}D\ [P(x, y)]) \to (\exists x{\in}D \forall y{\in}D\ [P(x, y)])$.

The formula within statement (iii) is not true, which can be seen by taking the interpretation $D := \mathbb{N}$ and $P(x, y) := x{>}y$. Under this interpretation, $\forall y{\in}D \exists x{\in}D\ [P(x, y)]$ is true, because for every $y \in \mathbb{N}$ we can indeed find a larger $x \in \mathbb{N}$, but $\exists x{\in}D \forall y{\in}D\ [P(x, y)]$ is not true, because there is no biggest number $x \in \mathbb{N}$. So the implication is false.

Definition 2.13
Suppose f and g are two formulas of predicate logic. We say that g follows from f, denoted as $f \models g$, when $\models f \to g$. Which means that in every situation in which f is true, g is true as well.

Statement (ii) above tells us that $\forall y{\in}D \exists x{\in}D\ [P(x, y)]$ follows from $\exists x{\in}D \forall y{\in}D\ [P(x, y)]$.

Definition 2.14
We say that formulas f and g are logically equivalent if the formula $f \leftrightarrow g$ is logically true. Using mathematical notation that would be: $f \equiv g$ when $\models f \leftrightarrow g$.

Example 2.15
Just like we had laws of *De Morgan* in propositional logic, we also have those in predicate logic. Now they define how to distribute negations over quantifiers. Let D be a non-empty domain and let P be a relation symbol with arity one. Then the following statements hold:
(i) $\neg\forall x \in D\ P(x) \equiv \exists x \in D\ \neg P(x)$
(ii) $\neg\exists x \in D\ P(x) \equiv \forall x \in D\ \neg P(x)$

So in fact, using these laws, we could have simply defined $\exists x \in D\ f$ as an abbreviation of the formula $\neg(\forall x \in D\ \neg f)$, instead of having it as a separate construction in the language of predicate logic as we did in Definition 2.1.

2.4 The language of predicate logic with equality

One might want to formalize the sentence:

Sharon is intelligent; there is a man who pays attention to nobody else.

To be able to formalize this, we require an *equality relation*. Then, we can write:

$$I(s) \wedge \exists x{\in}M\ [A(x,s) \wedge \forall w{\in}W\ [A(x,w) \to w = s]] \tag{2.2}$$

For this to be regarded a correct formula of predicate logic, we have to add the equality sign to the formal definition of the language of predicate logic.

Definition 2.16
The language of *predicate logic with equality* is defined by adding to the standard predicate logic the binary relation "=". The interpretation of this relation is always taken to be "is equal to". We then also have to add the following rule to the Definition 2.1 which states which formulas exist:

- If x and y are variables, and a and b are constants, then $(x = y)$, $(x = a)$, $(a = x)$, and $(a = b)$ are formulas as well.

Note that this equality binds stronger than the quantifiers and the logical operators. And as these formulas are *formulas in predicate logic*, there is no need to add a binary relation "$\neq$" with the interpretation of "is not equal to" to our formal language definition. For this we simply use the fact that we already have a negation operator and hence we can simply write something like $\neg(x = y)$. However, for convenience, just like the square brackets are introduced for convenience, *informally* we allow $x \neq y$ as an abbreviation for $\neg(x = y)$. So in general, in your formulas you are allowed to use this $x \neq y$, but, if you have to write a formula according to the official grammar, you have to use the long form $\neg(x = y)$.

Remark 2.17
The language of predicate logic with equality is very well suited to make statements about the number of objects having certain properties, such as there being exactly one such object, or at least two, or at most three, different, etc.

Example 2.18
Consider the sentence

Only Sharon is nice.

There are several ways to express this in predicate logic with equality, using the interpretation I_{13} (we leave the structure implicit):

H	domain of all human beings
$N(x)$	x is nice

1. $N(s) \wedge \forall x \in H\ [N(x) \to x = s]$
   This is the default pattern. It translates back into 'Sharon is nice and each human being that is nice, has to be Sharon'.

2. $N(s) \wedge \forall x \in H\ [\neg(x = s) \to \neg N(x)]$
   This formula states 'Sharon is nice and all human beings that are not Sharon, are not nice'.

3. There are actually two versions of the next pattern:

   (a) $\forall x \in H\ [N(x) \leftrightarrow x = s]$
   This is a slightly shorter pattern, but it may be more difficult to understand. It basically states 'Being nice is being Sharon'. It may seem that this formula actually doesn't state that Sharon is nice, but it does. As Sharon is an element of all people, one of the x's that need to be checked will be s, and then, obviously $x = s$ is true, which leads to the conclusion that $N(s)$ is also true because of the equivalence.

   (b) $\forall x \in H\ [x = s \leftrightarrow N(x)]$
   The same idea, but in the opposite order.

4. $N(s) \wedge \neg\exists x \in H\ [N(x) \wedge \neg(x = s)]$
   A pattern with an existential quantifier. It states that 'Sharon is nice and no one else is nice'.

Exercise 2.K
Consider the interpretation I_{14}:

H	domain of all human beings
$F(x)$	x is female
$P(x, y)$	x is parent of y
$M(x, y)$	x is married to y

Formalize the following sentences into formulas of predicate logic with equality:

(i) *Everyone has exactly one mother.*

(ii) *Everybody has exactly two grandmothers.*

(iii) *Every married man has exactly one spouse.*

Exercise 2.L
Use the interpretation I_{14} of Exercise 2.K to formalize the following properties.
(i) $C(x, y)$: x and y have had a child together.
(ii) $B(x, y)$: x is a brother of y (take care: refer also to the next item).
(iii) $S(x, y)$: x is a step-sister to y.

Translate the following formulas back to English.
(iv) $\exists x{\in}H \forall y{\in}H\ P(x, y)$. And is this true?

(v)

$$\begin{array}{llll}
\forall z_1 \in H \forall z_2 \in H & [& \exists x{\in}H \exists y_1{\in}H \exists y_2{\in}H & [\\
& & & \quad P(x, y_1) \\
& & & \wedge \\
& & & \quad P(y_1, z_1) \\
& & & \wedge \\
& & & \quad P(x, y_2) \\
& & & \wedge \\
& & & \quad P(y_2, z_2) \\
& & &] \\
& & \rightarrow & \\
& & \quad \neg\left(\exists w{\in}H \left[P(z_1, w) \wedge P(z_2, w)\right]\right) & \\
&] & &
\end{array}$$

And is this true?

Exercise 2.M

Given the interpretation I_{15}:

D	$\mathbb{N}$
$A(x, y, z)$	$x + y = z$
$M(x, y, z)$	$x \cdot y = z$

Formalize the following:

(i) $x < y$.

(ii) $x \mid y$ (x divides y).

(iii) x is a prime number.

2.5 Important concepts

Chapter 3

Discrete mathematics

This chapter deals with a number of small subjects that we have collected under the title of *discrete mathematics*. Notable about all these subjects, and the reason we have grouped them under this title, is that they all deal with natural numbers: the number of vertices in a graph, the number of steps in a recursive computation, the number of ways to traverse a grid from one point to another, etc. We never have to deal with problems of continuity, with an infinite number of points between two objects.

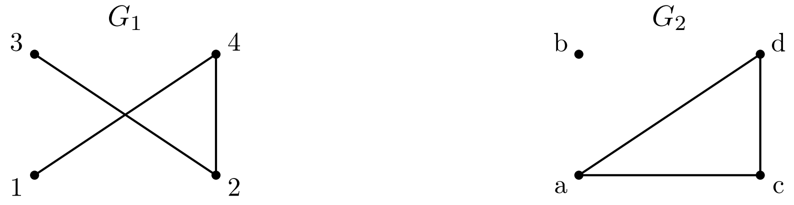
For more information, take a look at the introductory texts [1] or [5].

3.1 Graphs

This first section will give a short introduction to graph theory. Graphs are often encountered when studying things like languages, networks, data structures, electrical circuits, transport problems, flow diagrams, and so on.

Intuitively, a graph consists of a set V of vertices and a set E of edges between vertices.

Example 3.1
Two examples are:



All uncertainties you might have with this informal description, such as the questions: ‘When are two graphs the same?’, ‘Must the vertices lie on a flat surface?’, or ‘May the edges intersect each other?’, are resolved with a formal definition:

Definition 3.2
A *graph* is a tuple $\langle V, E\rangle$, of which V is a set of names, and E a set of 2-element subsets of V. The elements of V are called *vertices* (singular: *vertex*), or sometimes *nodes*, and the elements of E are called *edges*. We denote the edges not as usual sets, $\{v, w\}$, but instead as (v, w). So keep in mind that (v, w) and (w, v) denote the same edge.

Example 3.3
The graph G_1 from Example 3.1 is then $\langle V, E\rangle$, with $V = \{1, 2, 3, 4\}$ and $E = \{(1, 4), (2, 3), (2, 4)\}$. And $G_2 = \langle V, E\rangle$, with $V = \{a, b, c, d\}$ and $E = \{(a, c), (a, d), (c, d)\}$.

Remark 3.4
Notable in our Definition 3.2, is that V is allowed to be the empty set. Also, the edges don't have a *direction*: they are *lines* instead of *arrows*. Technically then, we are dealing with *undirected* graphs, where the automata of Chapter 5 are so-called directed graphs, with arrow edges.

Another remark is that it is technically impossible for an edge to connect a vertex to itself, because such an edge, connecting say v to itself, would be $(v, v) = \{v, v\} = \{v\}$, but that is a 1-element subset of V, and thus excluded by definition. Also impossible are two edges connecting the same two vertices. Of course, all these impossibilities could be resolved by changing the definition, but as it happens, this definition is elegantly simple, and already leads to enough mathematical expressivity and contemplation.

Definition 3.5
Let $G = \langle V, E\rangle$ be a graph, and $v, w \in V$.

- A *neighbor* of v is any vertex x for which $(v, x) \in E$.
- The *degree* (or: *valency*) of v is the number of neighbors v has.
- A *path* from v to w is a sequence of *distinct* edges $(x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n)$, with $n > 0$, $x_0 = v$, and $x_n = w$. We will also denote such a path as: $x_0 \to x_1 \to x_2 \to \cdots \to x_n$.
- With the statement 'G is *connected*' we mean to say that every pair of vertices has a path connecting the two.
- A graph $\langle V, E\rangle$ is a *sub-graph* of $\langle V', E'\rangle$ if $V \subseteq V'$ and $E \subseteq E'$.
- A *component* is an as large as possible connected sub-graph of G.
- A *cycle*, also sometimes called a *circuit*, is a path from a vertex to itself.
- With the statement 'G is *planar*', we mean to say that G can be drawn upon the plane, i.e. a flat surface, such that no edges intersect each other. (You are allowed to bend the edges if that is necessary.)
- With the statement 'G is a *tree*' we mean that G is connected and doesn't contain any cycles. (Which means, you can indeed picture it as a tree.)

To give an example of how real-world problems can be stated in terms of graph theory: the question whether a graph is planar is relevant to whether we can burn an electrical circuit onto a single layer chip.

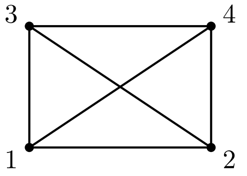
Exercise 3.A
Prove that in a tree, between any two points v and w, there is exactly one path that connects the two.

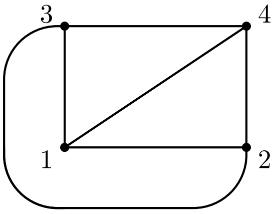
Exercise 3.B
A *bridge* in a graph G is an edge e for which: if you remove e from G, then the number of components of G increases. Prove that in a tree, every edge is a bridge.

Example 3.6
Let's take a look at some 'real-world' graphs.

(i) The '*country graph*' is $\langle V, E\rangle$, in which V is the set of all countries of the world, and E is the relation 'borders' (on the ground). The neighbors of the Netherlands are Germany and Belgium, and thus the degree of the Netherlands is 2. A path from the Netherlands to Spain would be, for example: the Netherlands → Germany → France → Spain. The country graph is not connected. $\{\text{England}, \text{Scotland}, \text{Wales}\}$ is a component. The Netherlands → Germany → Belgium → the Netherlands is a cycle. The country graph is planar.

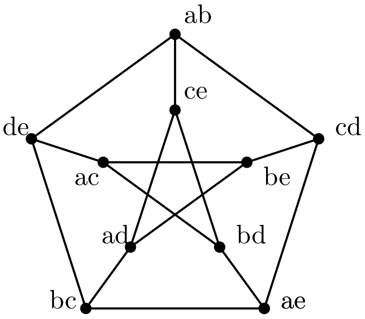
(ii) $K_4 = \langle\{1,2,3,4\}\ ,\ \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}\rangle$ (the *complete graph* with four points). Generally: K_n is the graph $\langle\{1,\ldots,n\}\, ,\, \{(v,w)\,|\, 1 \leq v < w \leq n\}\rangle$. The graph K_4 is planar. To understand this, note how instead of intuitively drawing it as the left picture, we can also draw it as in the right picture:





(iii) The *Petersen graph* is $\langle V, E\rangle$ where

$$\begin{cases} V = \{\text{ab}, \text{ac}, \text{ad}, \text{ae}, \text{bc}, \text{bd}, \text{be}, \text{cd}, \text{ce}, \text{de}\} \\ E = \{(v,w)\,|\, v \text{ and } w \text{ have no letters in common}\} \end{cases}$$



It can be shown that the Petersen graph is not planar. We will get back to that in Example 3.24.

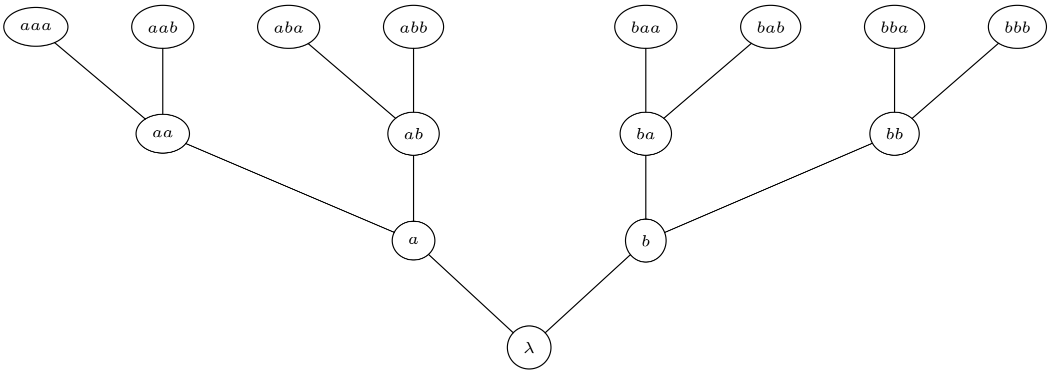
(iv) If a language is given by an inductive definition, we can make a corresponding graph. Take for example the language produced as follows:

axiom	λ
rule	$x \to xa$ $y \to yb$

We can then create the corresponding graph by:

- first, writing down all words that are included in the language, merely by an axiom (these are then the initial vertices in the graph),
- then, step-wise adding words (vertices) to the graph on the grounds of the languages' production rules, whereby we connect any new word to the word it was built up from (with its producing rule) with an edge.

For the above language, we then get this (unfinished, infinite) graph:

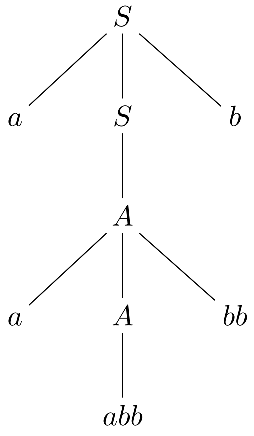


The graph is indeed a tree.

(v) For a word that is produced by a context-free grammar, you can create a *parse tree*, that demonstrates how the word is produced from the grammar. We have already seen such an example in Remark 4.17, but here we give a different one. Take the context-free grammar from Exercise 4.H:

$$\begin{aligned} S &\to aSb \mid A \mid \lambda \\ A &\to aAbb \mid abb \end{aligned}$$

The word $aaabbbbb$ is produced by this grammar, in the following way: $S \to aSb \to aAb \to aaAbbb \to aaabbbbb$. Below is the parse tree corresponding to this production.



In the parse tree, a leaf corresponds to a word, and a non-leaf vertex corresponds to a nonterminal that produces the word parts depicted by the vertices drawn below it. So, following the production rules $S \to aSb$, we draw a vertex with label S, and three subvertices: a leaf with label a, a non-leaf with label S, and a leaf with label b. The next production step, $A \to aAbb$, continues the graph generation, until we have ended up with the above. Reading the leaves, beginning at the top left, counter-clockwise around the graph, we get the produced word: $aaabbbbb$.

Remark 3.7
Information scientists almost always draw trees 'upside down': with the 'trunk' at the top and the branches pointing down, whereas mathematicians often draw trees with the right side up (see previous example).

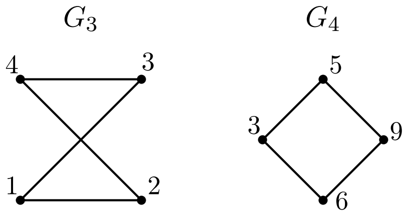
Exercise 3.C
Prove that for all $n \geq 1$, it holds that K_n has exactly $\frac{1}{2}n\,(n-1)$ edges.

3.2 Isomorphic graphs

Definition 3.8
A function f from a set A to a set B is called a *bijection* if every element in B has exactly one original. Formally: for every $b \in B$, there exists some $a \in A$ such that $f(a) = b$ and for every $a' \in A$ it holds that if $f(a') = b$, then $a = a'$.

Definition 3.9
We say that two graphs $\langle V, E\rangle$ and $\langle V', E'\rangle$ are *isomorphic* if there is a bijection $\varphi : V \to V'$ such that for all $v, w \in V$, we have $(v, w) \in E$ if and only if $(\varphi(v), \varphi(w)) \in E'$. Put differently: two graphs are called isomorphic if, disregarding the labels of the vertices, they are the same. A 'property preserving' bijection φ like the above is called an *isomorphism.*

Example 3.10
Consider the following two graphs.

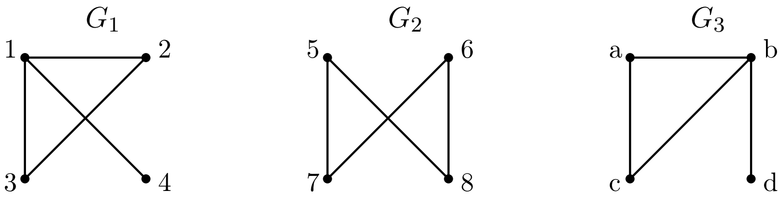


The graphs G_3 and G_4 are isomorphic, because we have an isomorphism φ:

$$\begin{aligned} 1 &\mapsto 6 \\ 2 &\mapsto 9 \\ 3 &\mapsto 3 \\ 4 &\mapsto 5 \end{aligned}$$

Isomorphic graphs are 'the same' if we are only interested in the graph-theoretic properties. For example: If G and G' are isomorphic and G is connected, then so is G'.

Exercise 3.D
Check which of the graphs below are isomorphic to each other:

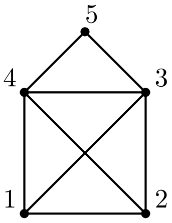


If two graphs are isomorphic, give an isomorphism between the two. If they are not, then explain why such an isomorphism cannot exist.

3.3 Euler and Hamilton

Definition 3.11
An *Eulerian path*, in a graph $\langle V, E\rangle$, is a path in which every edge from E is included exactly once. An *Eulerian circuit*, or *Eulerian cycle*, is an Eulerian path that is a cycle as well.

Example 3.12
Consider the graph:



The path 1 → 4 → 5 → 3 → 1 → 2 → 4 → 3 → 2 is an Eulerian path. Because vertex 1 has degree 3, this graph has no Eulerian circuit. This can be seen by examining the times that such a cycle would traverse vertex 1: if it does so only once, then one of its three edges cannot have been traversed, and if the cycle would traverse vertex 1 twice (or more), then at least one of its edges must have been used multiple times. So apparently, the graph doesn't admit an Eulerian cycle.

If we write out this argument in general, we have proved the following simple proposition:

Theorem 3.13 (Euler)
In a connected graph with at least two vertices:

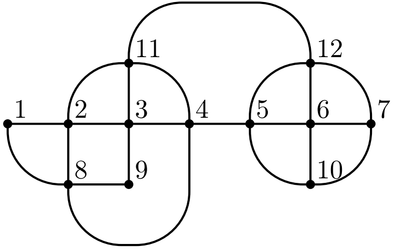
1. *An Eulerian circuit exists if and only if every vertex has an even degree.*
2. *An Eulerian path exists if and only if there are at most two vertices of odd degree.*

Eulerian circuits are of importance to for example newspaper deliverers, or to neighborhood police officers that want to efficiently pass through their neighborhood, ideally visiting each street exactly once and ending up where they started. A whole different problem is of importance to the 'traveling salesman', who wants to visit each of his customers (or cities) exactly once, returning home afterwards. He would be interested in a so-called 'Hamiltonian circuit':

Definition 3.14
A *Hamiltonian cycle*, or a *Hamiltonian circuit*, in a graph $\langle V, E\rangle$, is a cycle in which each vertex of V is traversed exactly once. A *Hamiltonian path*, in a graph $\langle V, E\rangle$, is a path in which each vertex of V is traversed exactly once, which is not also a cycle.

Remark 3.15
Whereas each Eulerian cycle is also an Eulerian path, this doesn't hold for Hamiltonian cycles, as Hamiltonian paths are by definition not cycles.[1] However, each Hamiltonian cycle can be reduced to a Hamiltonian path by removing a single, arbitrary, edge.

Exercise 3.E
Given here is a city map G of a village, on which streets are indicated by edges. There are pubs located on every vertex. The pubs are indicated by vertices, numbered 1 through 12:

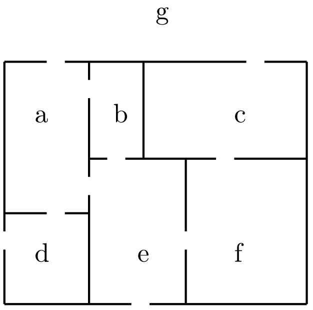


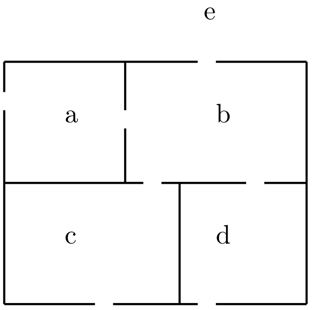
Formulate the following questions in terms of Hamiltonian and Eulerian circuits and paths, and answer them as well:

[1] This is a somewhat arbitrary decision within this course. There are also text books that do regard Hamiltonian cycles as Hamiltonian paths.

(i) Is it possible to make a walk in such a way that every street is traversed only once? If so, give an example, and if not, explain why.
(ii) Is it possible to make a walk, passing every street exactly once, and starting and ending in pub 3? If so, give an example, and if not, explain why.
(iii) Can a pub crawl be organized such that every pub is visited exactly once? If so, give an example, and if not, explain why.

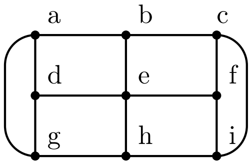
Exercise 3.F
Below, two floor plans of houses are given, in which the rooms and the garden have been given names.

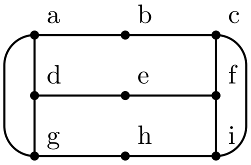




(i) For both floor plans, draw a corresponding graph, where the rooms, including the garden, become vertices, and the doors connecting rooms become edges connecting vertices.
(ii) For both houses, check whether it is possible to make a stroll through the house in such a way that every door is used exactly once, and you end up in the room where you started out. Explain your answer, and if you argue that such a stroll is possible, give an explicit example.
(iii) For both houses, check whether a stroll exists that passes through each room, as well as the garden, exactly once, returning to the original room afterwards. Explain your answer and give concrete examples if these strolls indeed exist.

Exercise 3.G
Which of the following graphs has a Hamiltonian circuit? Provide such a circuit or explain why a Hamiltonian circuit cannot exist.





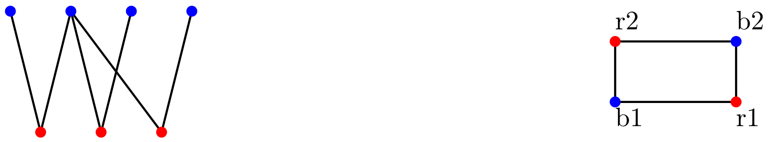
Exercise 3.H
Let $Q_3 = \langle V, E \rangle$ be the three dimensional hypercube graph, where V are the eight corners of a cube and E are the twelve edges connecting each vertex with three other vertices.
(i) Is Q_3 planar?
(ii) Does Q_3 have a Hamiltonian circuit?
(iii) Does Q_3 have an Eulerian circuit?
Don't forget to explain your answers!

Exercise 3.I
Show that the Petersen graph does contain a Hamiltonian path, but doesn't contain a Hamiltonian cycle.

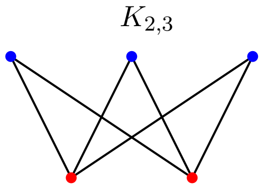
3.4 Colorings

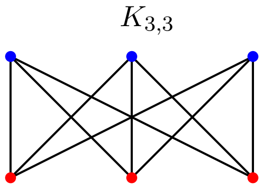
Definition 3.16
A graph $\langle V, E\rangle$ is called *bipartite* if V can be written as $V_1 \cup V_2$ where V_1 and V_2 are disjoint, in such a way that every edge leaving from a vertex of V_1, leads to a vertex of V_2, and the other way around. Put differently: you can color the vertices red and blue in such a way that no edge connects two vertices of the same color.

Example 3.17
Two examples of bipartite graphs:



Example 3.18
The *complete bipartite graph* with m red and n blue vertices, where every red vertex is connected to every blue vertex and the other way around, is denoted by $K_{m,n}$.

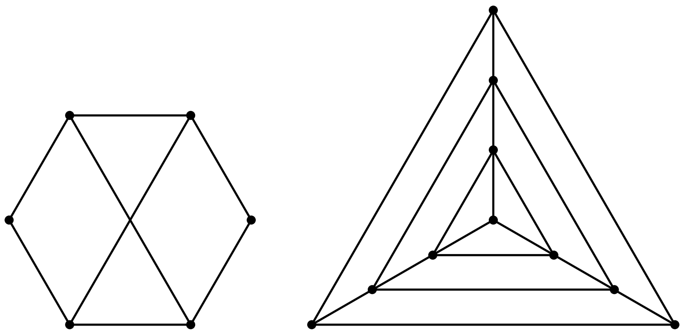




Definition 3.19
A *vertex coloring* of a graph $\langle V, E\rangle$ is a function $f : V \to \{1, \ldots, n\}$, such that for every edge (v, w), we have $f(v) \neq f(w)$. So: each vertex is assigned one of n colors, and neighboring vertices never share the same color.
The *chromatic number* of a graph is the least number n for which such a coloring is possible.

Example 3.20
Bipartite graphs are then graphs whose chromatic number is either 1 or 2.

Exercise 3.J
Find the chromatic number for the following two graphs. Explain your answer.



Exercise 3.K
Show that if a bipartite graph has a Hamiltonian path, the number of red vertices and blue vertices differs at most one.

Exercise 3.L
At a certain university quite a lot of language courses are being offered: Arabic, Bengali, Catalan, Danish, Estonian, Filipino and Greek. When creating the schedule for these courses the schedule maker has to take these requirements into account:

- All languages are being taught each day.
- Each lesson takes 105 minutes.
- The slots for the lessons are 08.45–10.30, 10.45–12.30, 13.45–15.30, 15.45–17.30 and 18.45–20.30.
- The building with five lecture rooms can only be rented as a whole, so the more courses are being taught in parallel, the cheaper it will be for the university.
- Some students have registered for more than one course and hence these courses should not be given in parallel. In the table below the places marked with $*$ indicate that there is at least one student that has registered for both the language in this row as the language in this column.

	A	B	C	D	E	F	G
A		$*$	$*$	$*$			$*$
B	$*$		$*$	$*$	$*$		$*$
C	$*$	$*$		$*$		$*$	
D	$*$	$*$	$*$			$*$	
E		$*$					
F			$*$	$*$			$*$
G	$*$	$*$				$*$	

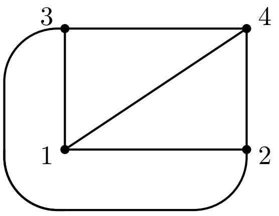
Give a schedule for the daily lessons that complies with the given requirements. Use graph theory to prove that your schedule is optimal.

We end our discussion about graphs with a few interesting theorems about planar graphs ans colorings. The proofs are typically too complicated for this course.

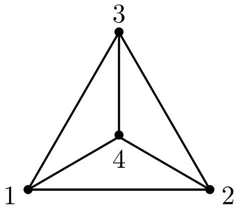
Theorem 3.21 (Fáry's theorem)
Every planar graph can be drawn with straight lines.

This theorem is known as Fáry's theorem, but it was actually already proven by Wagner in 1936, whereas Fáry proved it in 1948.

Example 3.22
In Example 3.6 we defined K_4 and we showed that it was a planar graph, by drawing it like this:



However, according to Theorem 3.21 we should be able to draw it without crossings using only straight lines. It is not difficult to see that this is indeed possible:

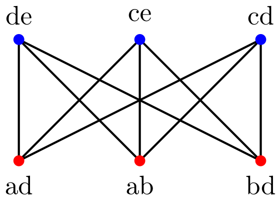


There are two important theorems that can be used to check whether a graph is planar or not.

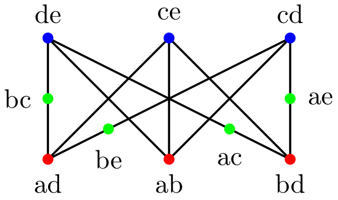
Theorem 3.23 (Kuratowski's theorem)
A graph is planar if and only if it contains no sub-graph homeomorphic *to K_5 or $K_{3,3}$.*

Here, two graphs G_1 and G_2 are homeomorphic if both can be obtained from the same graph G by adding new vertices of degree 2 in G by splitting existing edges.

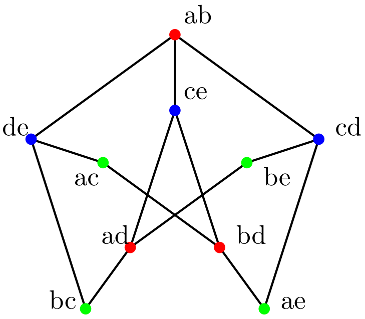
Example 3.24
Using Kuratowski's theorem we can show that the Petersen graph defined in Example 3.6 is not planar, as it has a sub-graph that is homeomorphic to $K_{3,3}$. (Even though the Petersen graph looks more like K_5, it doesn't have a sub-graph that is homeomorphic to K_5 as that would require vertices of degree 4, which the Petersen graph doesn't have.) So let us start with a graph G_1 which is clearly isomorphic to the graph $K_{3,3}$ as shown in Example 3.18, as the only difference is the name of the labels:



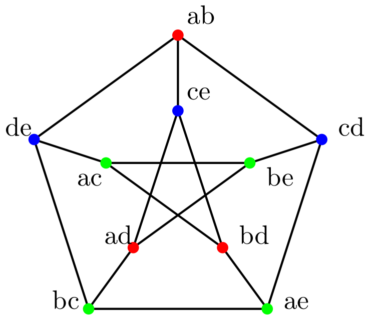
We will now create a graph G_2 which is homeomorphic to the graph G_1, by splitting the edges (ad,de), (ad,cd), (bc,de), and (bc,cd) and adding new vertices bc, be, ac, and ae respectively:



Now we can draw this same graph G_2 in a Petersen style and we get the following graph:



Now if we add the edges (bc,ae) and (ac,be) we get exactly the Petersen graph.



Hence we have shown that the Petersen graph has a sub-graph G_2, which is homeomorphic to G_1 which is isomorphic to $K_{3,3}$. So according to Kuratowski's theorem, the Petersen graph is not planar.

Remark 3.25
Note that the coloring of the vertices in the Petersen graph in Example 3.24 is not a coloring according Definition 3.19, as the adjacent vertices bc and ae are both green.

In Theorem 3.13 we have seen that Euler provided a simple numerical check for having Eulerian paths and circuits. He also has a simple check for showing that a graph is not planar:

Theorem 3.26 (Euler's formula)
Let G be a connected planar graph with n vertices and m edges, such that $n \geq 3$. Then the following statements hold:

- $m \leq 3n - 6$
- *if G does not have K_3 as a sub-graph, then $m \leq 2n - 4$.*

Note that not having K_3 as a sub-graph is a difficult way of saying that the graph contains no triangles.

Example 3.27
The Petersen graph has ten vertices and fifteen edges. So if we check the first criterion we get $15 \leq 3 \cdot 10 - 6 = 24$, which holds. Now as the Petersen graph does not contain triangles, we can also check the second criterion, leading to $15 \leq 2 \cdot 10 - 4 = 16$ which also holds. Hence as we already know that the Petersen graph is not planar, this example shows that Euler's formula provides *necessary* conditions for being planar, but that these conditions are not *sufficient*.

An interesting theorem by Appel and Haken, the proof of which was only found in 1975 and which is unfortunately too complicated to be included in this course, is the following:

Theorem 3.28 (Four color theorem)
The chromatic number of any planar graph is at most 4. Put differently: every map (of countries or cities, for example) can be colored using at most four colors, such that no adjacent territories share the same color.

3.5 Tower of Hanoi

The puzzle of the 'Tower of Hanoi' consists of three rods, on which a number of discs of different sizes are slid. The aim of the puzzle is to reconfigure the discs so that they all reside on one of the other rods, by subsequently moving a single disc from the top of one rod to another, but where it is not allowed to place a larger disc above a smaller disc.

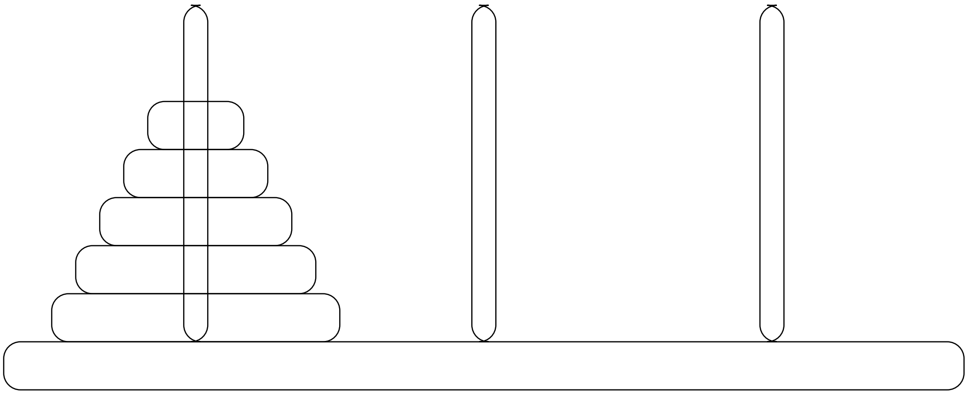


Figure 3.1: The tower of Hanoi with five discs. Can you solve the puzzle?

After some puzzling, you might succeed in solving the puzzle in Figure 3.1. But often, the strategy by which you did, is forgotten afterwards; moreover, we would like to solve the problem in general, for say 7 discs, or 8 discs, as well.

The important concept here is *generalizing* the problem: can we solve the puzzle for any number of discs? Note that the problem is very similar if we would only have four discs, and that the four disc problem, in turn, is very similar to the three disc problem.

Suppose we have already figured out how to solve the problem for four discs. Then, disregarding a fifth disc below all others on the initial rod, we can first transfer the four top discs to the second rod. Now, the fifth, large, disc, is cleared free on the first rod, and thus we move it to the third rod. Finally, we transfer the four discs unto it on the third rod, which we were able to do by assumption, resulting in all five discs now residing in order on the third rod, and we have solved the problem for five discs.

So, as you see, we can subsequently solve the problem for an extra disc, if we have already solved it for some number of discs.

Remark 3.29
This means that *if* we can solve the problem for a single disc, we can solve it for any number of discs, including the case of five discs, but also, say, ten or fifteen. And of course, the single disc problem is trivial: you can just move the disc freely.

Definition 3.30
The method we just demonstrated of solving a mathematical problem, is the method of *recursion*. Using recursion, one tackles a problem by dividing it into simpler problems, which are basically the same, ensuring that the solution of the larger problem then follows from those of the smaller problems. And of course, you should not forget to manually solve the smallest problem as well.

Remark 3.31
In general, the simple problem in recursion refers to the 'case 0' and the complex problem refers to the 'case $k+1$', where the k can be any natural number and the plus one ensures that $k+1$ is never zero. So the cases are really distinct.

In Remark 3.29, we stated that if we can solve the single disc problem, we can solve it for any number. However, in this case, we can actually go to an even simpler case: the case with zero discs! If we can solve this, then we can solve the problem for any number of discs using the strategy described above. And note that the problem is really trivial for zero discs, as nothing has to be done. In Figure 3.2 a Python implementation of our strategy is provided. Clearly, the 'if' branch refers to the 'case 0' and the 'else' branch to the 'case $k+1$'.

```
import sys

def hanoi (n, From, To, Via):
    if n==0:
        return
    else:
        hanoi(n-1,From,Via,To)
        print("move␣disk",n,"from",From,"to", To)
        hanoi(n-1,Via,To,From)

hanoi(int(sys.argv[1]),"left","right","middle")
```

Figure 3.2: Tower of Hanoi solution in Python

Now consider this question: given our strategy to solving the puzzle of Hanoi as described above, how many individual steps are needed? Call this number a_n, if we are dealing with the version with n discs. Of course, $a_0 = 0$. And it is also easy to see that $a_1 = 1$ and $a_2 = 3$. But what about a_5? It can be hard to see this at once. But again, recursion comes to the aid: in order to solve Hanoi with five discs, we would first solve it for four discs, then move the largest one to the third rod, and then use the four disc solution once more. In a formula, that would translate to: $a_5 = a_4 + 1 + a_4 = 2a_4 + 1$. Of course, this *recursive formula* is not specific for a_5, it holds in general: $a_{n+1} = 2a_n + 1$, for the numbers of discs $n \geq 0$. So, we can compute $a_3 = 2 \cdot a_2 + 1 = 2 \cdot 3 + 1 = 7$, and $a_4 = 2a_3 + 1 = 15$, and $a_5 = 2a_4 + 1 = 31$, etc... This gives us a neat sequence: $1, 3, 7, 15, 31, 63, 127, \ldots$

Exercise 3.M
The sequence a_n is recursively defined by:

$$\begin{array}{rcl} a_0 & = & 3 \\ a_{n+1} & = & a_n^2 - 2a_n \text{ for } n \geq 0 \end{array}$$

Use this definition to compute the value of a_5.

Exercise 3.N
The sequence b_n is recursively defined by:

$$\begin{array}{rcl} b_0 & = & 4 \\ b_{n+1} & = & b_n^2 - 2b_n \text{ for } n \geq 0 \end{array}$$

Use this definition to compute the value of b_5.

Exercise 3.O
Consider the sequence c_n for $n \geq 0$, given by the values:

$$1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, 521, 843, 1364, 2207, 3571, 5778, 9349, \ldots$$

Give a recursive definition for c_n.

3.6 Programming by recursion

Recursion is not only useful for doing mathematics, it is also an invaluable programming technique.

Suppose you have a little library in your programming language, that has an addition operation, however, it doesn't yet support multiplication. How would you define multiplication? With recursion, separating the 'case 0' and the 'case $k + 1$' again, that is easy:

$$\begin{array}{rcl} m * 0 & := & m \\ m * (k + 1) & := & m * k + m \end{array}$$

And once we have multiplication, we can also define exponentiation:

$$\begin{aligned} m^0 &:= 1 \\ m^{(k+1)} &:= m^k * m \end{aligned}$$

Both cases display how recursion allows you to define the operation (or, solve the problem) in terms of smaller cases, which are combined to form the new case.

Example 3.32
Although we talk of programming, the formulas above might not resemble actual computer code enough to bring home the point. Therefore, we added the following piece of Python code in Figure 3.3 for illustration purposes—and of course most other languages would allow a similar definition.

```
import sys;

def mult(m, n):
    """This is a recursive function to compute
       the product of m and n"""

    if n == 0:
        return 0
    else:
        return mult(m, n-1) + m


def power(m, n):
    """This is a recursive function to compute
       the value of m to the power n"""

    if n == 0:
        return 1
    else:
        return mult(power(m, n-1), m)


m = int(sys.argv[1])
n = int(sys.argv[2])
print("The␣product␣of␣" + str(m) + "␣and␣" + str(n) +
      "␣is␣" + str(mult(m,n)) + ".")
print("The␣value␣of␣" + str(m) + "␣to␣the␣power␣" + str(n) +
      "␣is␣" + str(power(m,n)) + ".")
```

Figure 3.3: Recursive Python procedures

Exercise 3.P
The Python program in Example 3.32 is not very robust. If we provide for n a negative integer, then Python will give the error *RecursionError: maximum recursion depth exceeded* as the base case is never reached. Now modify the program in such a way that if both m and n are integers, such that $|m| \leq 100$ and $|n| \leq 100$, Python gives the correct result.

Remark 3.33
In exercises and exams, we will sometimes ask for you to give a program using recursion. If we do so, so-called *pseudocode* will suffice: you need not actually know or adhere to a specific programming language. All is fine, as long as it is clear how the program works.

3.7 Binary trees

We come by another kind of recursive programming when studying what are called binary trees. Some examples of such a tree can be seen in Figure 3.4.

Although these trees might seem daunting, they were drawn with only a simple recursive procedure. The main insight here, is that any larger binary tree is composed of a stem, out of which two smaller binary trees grow: one going left, and one going right, both drawn slightly smaller, and of course at an angle.

So, a recursive recipe (or recursive function, or recursive procedure, or recursive program) is easily provided:

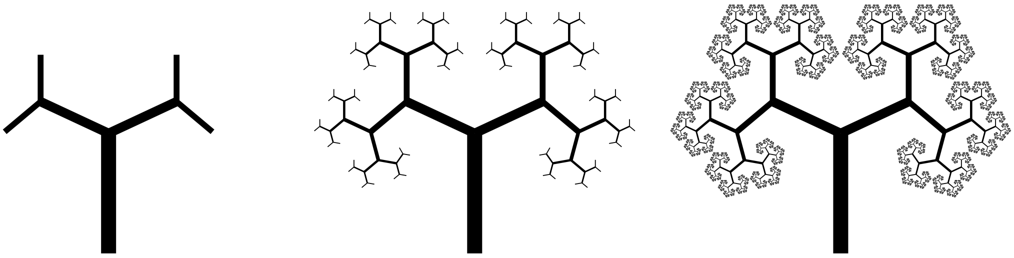


Figure 3.4: Binary trees

1. draw a stem
2. draw the left (sub)tree
3. draw the right (sub)tree

More precisely:

$$f(n,x,y,\alpha,\ell) = \begin{cases} \text{do nothing} & \text{if } n = 0 \\ 1.\ \textit{draw stem}(\text{position } \langle x,y\rangle, \text{angle } \alpha, \text{length } \ell) & \\ 2.\ f(n-1, \langle x',y'\rangle, \alpha - 65, \ell/1.6) & \text{if } n > 0 \\ 3.\ f(n-1, \langle x',y'\rangle, \alpha + 65, \ell/1.6) & \end{cases}$$

As input to the program are given: the height of the tree n, the starting position from which the tree should grow $\langle x, y\rangle$, the angle at which it should be drawn α, and the length its stem should have ℓ. The coordinates $\langle x', y'\rangle$ then denote the endpoint of its stem, and can be calculated from α and ℓ.

The height of the tree, n, is the most important input to this recursive procedure, for our purposes of explaining recursion. So in the list below we will omit the program's other variables.

The task of drawing a tree of height n is reduced to the task of drawing two trees of height $n - 1$, and hence this program indeed uses recursion. This can be clearly seen when taking a look at the steps that the computer will consecutively take to execute this program:

```
[1] draw tree: f(3)
[1.1] draw stem
[1.2] draw left tree: f(2)
[1.2.1] draw stem
[1.2.2] draw left tree: f(1)
[1.2.2.1] draw stem
[1.2.2.2] draw left tree: f(0)
[1.2.2.2.1] do nothing
[1.2.2.3] draw right tree: f(0)
[1.2.2.3.1] do nothing
[1.2.3] draw right tree: f(1)
[1.2.3.1] draw stem
[1.2.3.2] draw left tree: f(0)
[1.2.3.2.1] do nothing
[1.2.3.3] draw right tree: f(0)
[1.2.3.3.1] do nothing
```

```
[1.3] draw right tree: f(2)
[1.3.1] draw stem
[1.3.2] draw left tree: f(1)
[1.3.2.1] draw stem
[1.3.2.2] draw left tree: f(0)
[1.3.2.2.1] do nothing
[1.3.2.3] draw right tree: f(0)
[1.3.2.3.1] do nothing
[1.3.3] draw right tree: f(1)
[1.3.3.1] draw stem
[1.3.3.2] draw left tree: f(0)
[1.3.3.2.1] do nothing
[1.3.3.3] draw right tree: f(0)
[1.3.3.3.1] do nothing
```

The trees in Figure 3.4 are drawn with $n = 3$, $n = 7$, and $n = 15$.

Note how a *recursive definition* is often very simple, though the *execution* of a recursive function can be quite complicated: quite a lot of book-keeping can be required to keep track of which sub-problem is being solved at any given point in time. Typical work for computers!

3.8 Induction

It may also help if you understand how falling dominoes work. Under the assumption that they are close enough to each other, if the first one falls, then all dominoes fall. Again there is a 'case 0' and a 'case $k + 1$' domino. In 'case 0', the first domino falls because it is being pushed over. In 'case $k + 1$', each other domino falls because it is hit by the previous domino falling over. Let's return back to the sequence we encountered earlier, the sequence of the number of steps needed to solve the puzzle of Hanoi with n discs. This sequence was given by the recursive formula $a_{n+1} = 2a_n + 1$, where a_n stands for the number of steps needed. Of course computing a_n for a high n, for example a_{38}, can now be a bit tedious, because we have to first compute all previous a_n's. It would be more convenient if we had a direct formula to compute a_{38}.

Taking another look at the sequence: $1, 3, 7, 15, 31, 63, 127, \dots$, one might notice that it looks very much like the sequence of powers of two: $2, 4, 8, 16, 32, 64, 128, \dots$ The numbers of the first sequence seem to always be one less than those of the powers of two. But how do we know this for sure? Couldn't it be just a coincidence of the first part of the sequences? Suppose for now that $a_{37} = 2^{37} - 1$ would be indeed the case. Then we also have:

$$a_{38} = 2 \cdot a_{37} + 1 = 2 \cdot \left(2^{37} - 1\right) + 1 = 2^{38} - 2 + 1 = 2^{38} - 1.$$

So, if it holds for the 37th element, then also for the 38th, and so on. Of course there is nothing specific about it being exactly the 37th going on above, so we have the general truth that if $a_k = 2^k - 1$ for some k, then also $a_{k+1} = 2^{k+1} - 1$ for the same k. Furthermore, we already know that $a_0 = 2^0 - 1 = 0$. So then indeed, it holds for a_1, and then also for a_2, and for a_3, and so on, and in particular also for a_{38}. We have now proved that indeed for all n, we have $a_n = 2^n - 1$, and this is what we call a *direct formula* for a_n, because it doesn't depend on any previous values. Note that we used the distinction between the 'case 0' and the 'case $k + 1$' again!

The method of proof demonstrated above is what we call *induction*. As stated before, induction, as a proof technique, is very similar to recursion, a way to define recursive formulas.

Definition 3.34
Induction can be used to prove that a certain *predicate* $P(n)$ holds for all natural numbers $(0, 1, 2, 3, \dots)$. Such a *proof by induction* is given by:

1. A proof of $P(0)$. (The *base case.*)
2. A proof that: if $P(k)$ holds for some $k \in \mathbb{N}$, where k is greater than or equal to the base case, then also $P(k+1)$. (The *induction step.*)

Giving these two is then enough to show that $P(n)$ holds for all n. When proving $P(k+1)$, (in 2.), the assumption that $P(k)$ holds already is called the *induction hypothesis* (IH).

To see how, for example, $P(37)$ then follows from such a proof by induction, note first that $P(0)$ holds, and thus (by 2.) also $P(1)$, and thus (again by 2.) also $P(2)$, etc, until we arrive at the truth of $P(37)$.

In our example above, the predicate $P(n)$ was the statement that $a_n = 2^n - 1$.

Remark 3.35
A recurring question is whether one should start with $P(0)$ or with $P(1)$. The answer to this question depends on the situation. If you want to prove something about *all* natural numbers, then you should of course start with $P(0)$. But if you only want to prove something about all natural numbers greater than, say, 7, then of course you may start with $P(8)$. It may sometimes even be necessary to prove the first number of cases separately, because the regularity only arises after that. Your induction step proof might then be for, say, $k \geq 5$.

Remark 3.36
In a way, induction can be seen as the opposite of recursion. The emphasis of induction lies on consecutively proving larger cases, while recursion is used to break a problem down into smaller cases: the opposite of the first.

Example 3.37
How much does $1 + 2 + 3 + \cdots + 99$ add up to? You could calculate this manually, or use a calculator, but in both cases you would be sure to be busy for a while. Unless you are smart of course, and use recursion and induction. Suppose you define $s(n) := 1 + 2 + \cdots + n$, for $n \geq 1$. Then a recursive program for computing $s(n)$ would be $s(n+1) = s(n) + n + 1$.

After looking at the first number of results $s(1)$, $s(2)$, $s(3)$, ..., you might start to suspect that generally $s(n)$ is given by $s(n) = \frac{n^2+n}{2}$. To prove this direct formula for the value of $s(n)$, we use induction. In this example our induction predicate is defined by

$$P(n) := \quad \left[s(n) = \frac{n^2+n}{2} \right]$$

and we are going to prove that it holds for all $n \geq 1$.

Base Case Does $P(1)$ hold? Yes, because $s(1) = 1 = \frac{1^2+1}{2}$.

Induction Step Now suppose that we already know that $P(k)$ holds for some $k \in \mathbb{N}$ with $k \geq 1$, that is, $s(k) = \frac{k^2+k}{2}$ (IH). Do we have that $P(k+1)$ holds as well? More precisely, do we have that $s(k+1) = \frac{(k+1)^2+(k+1)}{2}$? Let's see what we can say about $s(k+1)$:

$$\begin{aligned}
s(k+1) &= s(k) + k + 1 \text{ (by definition of } s(k+1)) \\
&= \frac{k^2+k}{2} + k + 1 \text{ (IH)} \\
&= \frac{k^2+k+2k+2}{2} \text{ (algebra)} \\
&= \frac{(k^2+2k+1)+(k+1)}{2} \text{ (algebra)} \\
&= \frac{(k+1)^2+(k+1)}{2} \text{ (algebra)}
\end{aligned}$$

So $P(k+1)$ indeed holds.

So indeed, it follows by induction that $s(n) = \frac{n^2+n}{2}$ holds generally for $n \geq 1$.

The mathematician Gauss, by the way, was able to solve this problem in an easy way, without using induction. See Exercise 3.U.

Example 3.38
In how many ways can we order the numbers 1, 2, 3, 4, 5, 6, 7, 8 and 9? It is cumbersome to write down all possibilities. But luckily, we don't need to. Let a_n denote the number of orderings of n elements. So, the question is what the value of a_9 is. At least we know where to start: a_1 is simply 1. Furthermore, $a_{n+1} = (n+1) \cdot a_n$, because we would first choose an element to put at the front of the list, for which we have $n+1$ options, and then we order the remaining n elements, for which there are a_n possibilities by definition.

So how, then, do we calculate a_9? Well, the definition we gave above is exactly that of the factorial: $a_n = n!$. Which means, we can use any standard scientific calculator.

Example 3.39
What is the value of the sum

$$1 + a + a^2 + a^3 + \cdots + a^n$$

given some $n \in \mathbb{N}$ and an arbitrary a? Let's first test it out on some small values:

	$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 10$	$n = 42$
$a = 0$	1	1	1	1	1	1	1
$a = 1$	1	2	3	4	5	11	43
$a = 3$	1	4	13	40	121	88573	164128483697268538813
$a = \frac{2}{3}$	1	$\frac{5}{3}$	$\frac{19}{9}$	$\frac{65}{27}$	$\frac{211}{81}$	$\frac{175099}{59049}$	$\frac{328256958598444055419}{109418989131512359209}$
$a = -1$	1	0	1	0	1	1	1
$a = -2$	1	-1	3	-5	11	683	2932031007403

An attentive student might remember the formula for calculating the sum of a geometric sequence:

$$1 + a + a^2 + a^3 + \cdots + a^n = \begin{cases} n+1 & \text{if } a = 1 \\ \frac{a^{n+1}-1}{a-1} & \text{if } a \neq 1 \end{cases}$$

We will prove by induction that the case where $a \neq 1$ holds. We start off by defining precisely what our predicate $P(n)$ is:

$$P(n) := \quad \left[1 + a + a^2 + a^3 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \right]$$

Base Case We want to prove the statement for all $n \in \mathbb{N}$, so we start with the base case of $P(0)$, which means that we have to prove that:

$$1 = \frac{a^{0+1} - 1}{a - 1}$$

Indeed this is the case, because $a^{0+1} - 1 = a^1 - 1 = a - 1$. So the numerator and the denominator of the fraction are equal, which implies that the fraction is indeed equal to 1. Recall that we have $a \neq 1$ by assumption, so we don't have to worry about dividing by zero.

Induction Step We may assume the induction hypothesis, $P(k)$ for some $k \in \mathbb{N}$ such that $k \geq 0$, and must now prove that $P(k+1)$ holds as well, that is:

$$1 + a + a^2 + a^3 + \cdots + a^k + a^{k+1} = \frac{a^{(k+1)+1} - 1}{a - 1}$$

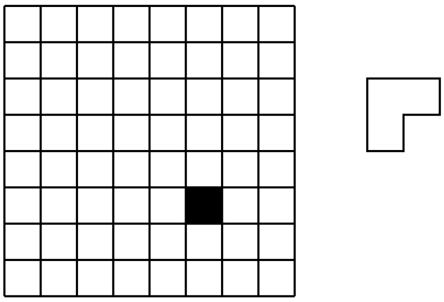
We do this by rewriting the left hand side of the equation until we see how our assumption $P(k)$ fits into it, then use the induction hypothesis, and then rewrite a bit more until we have the right hand side of the equation.

$$\begin{aligned}
&1 + a + a^2 + a^3 + \cdots + a^k + a^{k+1} \\
&= (1 + a + a^2 + a^3 + \cdots + a^k) + a^{k+1} \text{ (reveal left hand side of } P(k)) \\
&= \frac{a^{k+1} - 1}{a - 1} + a^{k+1} \text{ (apply IH)} \\
&= \frac{a^{k+1} - 1}{a - 1} + a^{k+1} \cdot \frac{a - 1}{a - 1} \text{ (making the denominators equal)} \\
&= \frac{a^{k+1} - 1}{a - 1} + \frac{a^{k+2} - a^{k+1}}{a - 1} \text{ (multiplying of fractions)} \\
&= \frac{a^{k+1} - 1 + a^{k+2} - a^{k+1}}{a - 1} \text{ (adding fractions)} \\
&= \frac{a^{k+2} - 1}{a - 1} \text{ (canceling terms in numerator)} \\
&= \frac{a^{(k+1)+1} - 1}{a - 1} \text{ (making it look exactly as in } P(k+1))
\end{aligned}$$

And now we have ended up with the right hand side of the equation of $P(k+1)$.

So now it follows by induction that $P(n)$ holds for all $n \in \mathbb{N}$ and $a \neq 1$.

Example 3.40
So far we have only concerned ourselves with theorems of which the proofs are based on, and rely upon skills of, algebra. This example demonstrates that other options are possible as well. Consider this 8×8 board, of which one of its squares has been removed. Can the rest of the board be tiled completely with only tiles of the particular shape pictured next to the board?



A bit of puzzling might convince you that this is indeed possible. But the question is pressing, whether this is coincidental to the particular square removed from the board, or whether it may be tiled whichever square is removed. The latter turns out to generally be the case, for which we will now give an inductive proof.

First, we define our predicate $P(n)$:

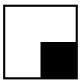
$P(n) \quad :=$ [A $2^n \times 2^n$ board of which a single square has been removed, can always be tiled (with tiles of the shape pictured above), regardless of which square was removed.]

Now we can make the theorem clear:

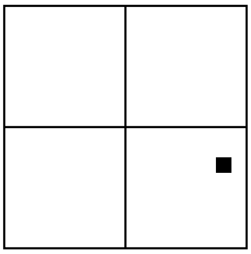
Theorem *The predicate $P(n)$ holds for all $n \geq 1$.*

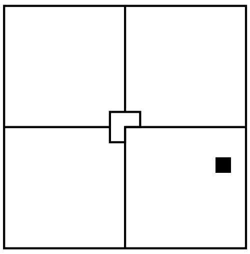
Proof by induction on n.

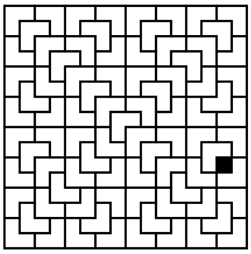
Base Case The base case is $P(1)$, because the theorem explicitly tells us that $n \geq 1$. Which means, we must prove that a 2×2 board can be tiled, if any single square has been removed. This is of course easily seen to be true. Whichever square is removed doesn't matter, because we can simply rotate the tile. The picture below illustrates the base case tiling.



Induction Step Let $k \in \mathbb{N}$ and $k \geq 1$. We may now assume that $P(k)$ holds (our induction hypothesis), which means that any $2^k \times 2^k$ board with one square removed can be tiled with the special tile. And we now have to prove that $P(k+1)$ holds, which means that any $2^{k+1} \times 2^{k+1}$ board with one square removed can be tiled. Now let us consider an arbitrary $2^{k+1} \times 2^{k+1}$ board, of which one square has been removed. Note that such a board can always be subdivided into four $2^k \times 2^k$ board, of which one of them has a square removed. By rotating our initial board, we can state without loss of generality, that the square will have been removed in the lower right sub-board. The leftmost illustration depicts this situation.







Now, crucially, we place the first tile in the center of our board, as depicted in the middle illustration. Note that now each sub-board can be regarded as having 'removed' a single square, except for the lower right board—but it already had a square removed. Now, we can simply apply our induction hypothesis to all four sub-boards, thereby immediately yielding a tiling of the full board as well. And so we have proved that $P(k+1)$ holds.

So with induction it follows that $P(n)$ holds for all $n \geq 1$.

Note that in this proof, we have not only proved that $P(n)$ holds generally, but we have actually specified a definite method for tiling. The rightmost illustration depicts the result of this tiling method for a 16×16 board.

Exercise 3.Q
Consider the sequence a_n defined by:

$$\begin{aligned} a_0 &= 0 \\ a_{n+1} &= a_n + 2n + 1 \quad \text{for } n \geq 0 \end{aligned}$$

Prove by induction that for all $n \in \mathbb{N}$, $a_n = n^2$.

Exercise 3.R

The inventor of the chessboard was told by the king of Persia, that he would be rewarded any object of choice. The inventor chose the following: 1 grain of rice on the first field of the chessboard, 2 grains of rice on the second, 4 on the third, and so on, doubling the number of grains for each successive field. The king thought the inventor to be very humble. Now the question is: how many grains of rice did the inventor's choice amount to? Let's formulate it formally: he asked for

$$1 + 2 + 2^2 + 2^3 + 2^4 + \cdots + 2^{63}$$

grains of rice. Can you find a direct formula for the result of this sum for a generalized board with n fields? And can you then prove this formula by induction? [*Hint:* This is what you could do here: Give a recursive definition for s_n, the number of rice grains on the first n fields. Make a table with three columns: one column for n, one for the expression $1 + 2 + \cdots + 2^{n-1}$ and one for s_n. Use this table to guess a direct formula $f(n)$ for s_n. Prove by induction that $f(n) = s_n$ for all $n \geq 1$. Use the direct formula to compute s_{64}.]

Exercise 3.S

In Exercise 3.C, we proved that for all $n \geq 1$ the complete graph K_n has exactly $\frac{1}{2}n(n-1)$ edges. Now, try to prove this by induction on n.

Exercise 3.T

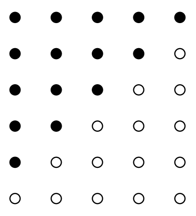
Prove by induction, that for all $n \in \mathbb{N}$, $2^n \geq n$.

Exercise 3.U

So we have seen the formula for the sum of the following sequence:

$$1 + 2 + 3 + 4 + 5 + \cdots + n = \frac{(n+1)\,n}{2}.$$

What is the connection with this picture?

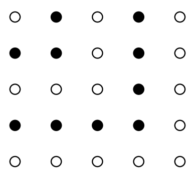


Exercise 3.V

How many distinct sequences of length n can we make, filled with the numbers 1 through 5? This too, can be solved easily with recursion. Let a_n denote the number of distinct sequences of length n. The simplest sequence, of length one, can of course be made in five ways, so $a_1 = 5$. A sequence of length $n+1$ can be made by first taking a sequence of length n, and then appending a new element after it, for which we have five options. So, $a_{n+1} := 5 \cdot a_n$. Now, recall the definition of raising to a power. Prove by induction, that for all $n \geq 1$, $a_n = 5^n$.

Exercise 3.W

Prove, by induction, that $1 + 3 + 5 + 7 + \cdots + (2n-1) = n^2$ for $n \geq 1$. What is the connection with the picture below?



3.9 Pascal's Triangle

Before we discuss Pascal's Triangle, we repeat some combinatorics by an example.

Example 3.41
We will be taking a look at the Dutch Lotto[2]. In this game of chance, there is a 'vase', filled with 45 sequentially numbered balls (1 up to 45). A notary then pulls six balls from this vase, at random (and without putting them back afterwards). The numbers on these six balls are then compared to the numbers that the players have chosen beforehand. The more numbers coincide, the higher the rewards are. But how many distinct outcomes are there to such a draw? Well, for the first ball to be drawn, there are of course 45 possibilities. And for the second, 44, and for the third, 43, etc. So that would lead us to the conclusion that there are

$$45 \cdot 44 \cdot 43 \cdot 42 \cdot 41 \cdot 40 = \frac{45 \cdot 44 \cdot 43 \cdot 42 \cdot 41 \cdot 40 \cdot 39 \cdot 38 \cdot \cdots \cdot 1}{39 \cdot 38 \cdot \cdots \cdot 1} = \frac{45!}{39!} = \frac{45!}{(45-6)!}$$

ways to draw six consecutive balls. But note that the ordering of these balls, after they have been drawn, doesn't make a difference. So for example, the draw $(5, 10, 2, 29, 6)$ is equal, with respect to this game, to the draw $(10, 2, 5, 29, 6)$, but these two have both been counted once in the calculation above. So we have to compensate for all draws that we counted 'double'. Each ordering of six of these balls leads to the same outcome. We know that the number of orderings of six of these balls is 6!, as we have seen earlier, so to counter this problem, we must divide the result of our calculation above by 6!, giving:

$$\frac{45!}{6!\,(45-6)!} = 8145060$$

This is then the actual number of possible draws.

Definition 3.42
Let n and k be natural numbers, with $k \leq n$. We then define the *binomial* $\binom{n}{k}$ as the number of ways in which k objects can be drawn from a set of n elements (as above). This number can be computed using this formula:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

It is pronounced as 'n choose k'.

We now continue by looking at the grid coordinates (n, k) of a grid that is skewed in such a way that we have $(0, 0)$ at the top:

$$\begin{array}{c}
(0,0) \\
(1,0) \quad (1,1) \\
(2,0) \quad (2,1) \quad (2,2) \\
(3,0) \quad (3,1) \quad (3,2) \quad (3,3) \\
(4,0) \quad (4,1) \quad (4,2) \quad (4,3) \quad (4,4) \\
(5,0) \quad (5,1) \quad (5,2) \quad (5,3) \quad (5,4) \quad (5,5) \\
(6,0) \quad (6,1) \quad (6,2) \quad (6,3) \quad (6,4) \quad (6,5) \quad (6,6) \\
(7,0) \quad (7,1) \quad (7,2) \quad (7,3) \quad (7,4) \quad (7,5) \quad (7,6) \quad (7,7)
\end{array}$$

We will now assign values to these coordinates in different ways, and see how these different assignments relate.

[2] Although we will disregard 'Superzaterdag' and the 'Jackpot' for convenience.

Definition 3.43
Assign ones to the left and right borders of the triangle, and then fill in the rest of the triangle by consecutively adding up the two values directly above a new one. Put more precisely: all coordinates $(n, 0)$ and (n, n) get the value 1, and then each (n, k) is given by adding up the values of $(n-1, k)$ and $(n-1, k-1)$. *The first version of Pascal's Triangle* ($\text{P}\triangle_1$) is thus as follows[3]:

$$\begin{array}{c}
1 \\
1 \quad 1 \\
1 \quad 2 \quad 1 \\
1 \quad 3 \quad 3 \quad 1 \\
1 \quad 4 \quad 6 \quad 4 \quad 1 \\
1 \quad 5 \quad 10 \quad 10 \quad 5 \quad 1 \\
1 \quad 6 \quad 15 \quad 20 \quad 15 \quad 6 \quad 1 \\
1 \quad 7 \quad 21 \quad 35 \quad 35 \quad 21 \quad 7 \quad 1
\end{array}$$

Definition 3.44
Assign to each coordinate (n, k), the value $\binom{n}{k}$ as defined in Definition 3.42. This then gives us *the second version of Pascal's Triangle* ($\text{P}\triangle_2$).

The numbers on the outer edge of $\text{P}\triangle_2$ all have coordinates either $(n, 0)$ or (n, n). By definition of $\text{P}\triangle_2$, that means that the assigned number is either $\binom{n}{0}$ or $\binom{n}{n}$, respectively, both of which compute to 1, whatever n is. So all numbers on the outer edge of $\text{P}\triangle_2$ are 1.

In addition, we claim, for $k < n$, that $\binom{n+1}{k+1} = \binom{n}{k+1} + \binom{n}{k}$. If we draw a subset A containing $k+1$ elements from the set $1, 2, 3, 4, \ldots n+1$, then there are two possibilities: either $n+1 \in A$, or $n+1 \notin A$. In the first case, $A \setminus \{n+1\}$ is a subset containing k elements, drawn from $\{1, 2, 3, 4, \ldots, n\}$. In the second case, A is a subset containing $k+1$ elements, drawn from $\{1, 2, 3, 4, \ldots, n\}$. So indeed $\binom{n+1}{k+1} = \binom{n}{k+1} + \binom{n}{k}$. Or, put differently: $\text{P}\triangle_2$ also follows the principle that the value of each coordinate can be found by adding the values of the two coordinates that lie above it. The result of which is that the two triangles $\text{P}\triangle_1$ and $\text{P}\triangle_2$ are exactly the same.

In Definition 3.42 we have defined the binomial coefficient in a combinatorial way and by giving a direct formula. However, as we have seen that the binomial coefficients are the elements in the triangle of Pascal, we can also give a recursive definition:

Definition 3.45
Let n and k be natural numbers. The *binomial coefficients* $\binom{n}{k}$ are defined by these four equations:

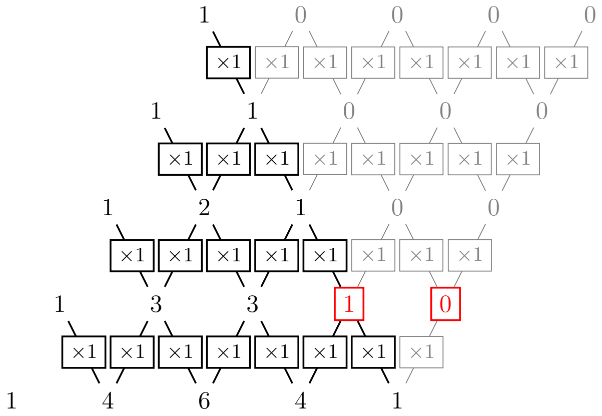
$$\begin{array}{llll}
1. & \dbinom{0}{0} = 1 & 2. & \dbinom{0}{k+1} = 0 \\
3. & \dbinom{n+1}{0} = 1 & 4. & \dbinom{n+1}{k+1} = \dbinom{n}{k} + \dbinom{n}{k+1}
\end{array}$$

This definition is slightly different from what you can find in other sources like Wikipedia[4], where there usually is a specific definition for $\binom{n}{n} = 1$. In particular, this definition works for all natural numbers n and k, even if $k > n$. This is due to the $\binom{0}{k+1} = 0$ equation, which

[3] Of course this triangle carries on infinitely, we just abbreviated it to the first eight rows.

[4] `https://en.wikipedia.org/wiki/Binomial_coefficient`

explicitly states that to the right of the $\binom{0}{0}$ at the top of Pascal's triangle there is an infinite series of zeros, which are not shown in the triangle itself. This row of zeros allows for the computation of $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$ for any n and k. For instance, if we want to compute $\binom{3}{3}$ (which is on the border of the triangle) and $\binom{3}{4}$ (which is just outside the triangle), the computations go like this:



$$\begin{aligned}
\binom{3}{3} &= \binom{2+1}{2+1} && \text{algebra}\\
&= \binom{2}{2} + \binom{2}{2+1} && \text{eq. 4}\\
&= \binom{1+1}{1+1} + \binom{1+1}{2+1} && \text{algebra}\\
&= \binom{1}{1} + \binom{1}{1+1} + \binom{1}{2} + \binom{1}{2+1} && 2\times \text{ eq. 4,}\\
&= \binom{0+1}{0+1} + \binom{0+1}{1+1} + \binom{0+1}{1+1} + \binom{0+1}{2+1} && \text{algebra}\\
&= \binom{0}{0} + \binom{0}{0+1} + \binom{0}{1} + \binom{0}{1+1} + \binom{0}{1} + \binom{0}{1+1} + \binom{0}{2} + \binom{0}{2+1} && 4\times \text{ eq. 4}\\
&= 1 + 0 + \binom{0}{0+1} + 0 + \binom{0}{0+1} + 0 + \binom{0}{1+1} + 0 && \text{algebra, eq. 1, } 4\times \text{ eq. 2}\\
&= 1 + 0 + 0 + 0 + 0 + 0 + 0 + 0 && 3\times \text{ eq. 2}\\
&= 1 && \text{algebra}
\end{aligned}$$

$$\begin{aligned}
\binom{3}{4} &= \binom{2+1}{3+1} && \text{algebra}\\
&= \binom{2}{3} + \binom{2}{3+1} && \text{eq. 4}\\
&= \binom{1+1}{2+1} + \binom{1+1}{3+1} && \text{algebra}\\
&= \binom{1}{2} + \binom{1}{2+1} + \binom{1}{3} + \binom{1}{3+1} && 2\times \text{ eq. 4}\\
&= \binom{0+1}{1+1} + \binom{0+1}{2+1} + \binom{0+1}{2+1} + \binom{0+1}{3+1} && \text{algebra}\\
&= \binom{0}{1} + \binom{0}{1+1} + \binom{0}{2} + \binom{0}{2+1} + \binom{0}{2} + \binom{0}{2+1} + \binom{0}{3} + \binom{0}{3+1} && 4\times \text{ eq. 4}\\
&= \binom{0}{0+1} + 0 + \binom{0}{1+1} + 0 + \binom{0}{1+1} + 0 + \binom{0}{2+1} + 0 && \text{algebra, } 4\times \text{ eq. 2}\\
&= 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 && 4\times \text{ eq. 2}\\
&= 0 && \text{algebra}
\end{aligned}$$

Note that we did a lot of explicit rewriting like $\binom{3}{4} = \binom{2+1}{3+1}$. This is needed, because $\binom{3}{4}$ does not match any of the four definitions, where $\binom{2+1}{3+1}$ does match with $\binom{n+1}{k+1}$ for $n = 2$ and $k = 3$.

Exercise 3.X
We have seen above that $\binom{3}{3} = 1$ and $\binom{3}{4} = 0$. Now use Definition 3.45 to prove the following propositions by induction on n:

(i)

$$\text{for all } l \text{ such that } l > n \text{ it holds that } \binom{n}{l} = 0 \text{ for all } n \geq 0$$

(We use l as k is already needed in the induction scheme.)

(ii)

$$\binom{n}{n} = 1 \text{ for all } n \geq 0$$

[*Hint:* You may need the result of the previous proposition.]

Definition 3.46
Assign to each coordinate (n, k), the number that denotes the number of possible roads leading from $(0, 0)$ to (n, k), where each subsequent step should be one step down, either diagonally to the left, or to the right. This gives us *the third version of Pascal's Triangle* (P$\triangle_3$).

One can then observe that:

- The border of P$\triangle_3$ is filled with ones.
- The triangle P$\triangle_3$ also follows the 'principle of adding' described above for the other two versions of Pascal's Triangle.

Result: P$\triangle_3$ is yet again exactly the same triangle as P$\triangle_1$ and P$\triangle_2$.

Definition 3.47
Assign to each coordinate (n, k), the coefficient that x^k takes on in the polynomial $(1 + x)^n$. Example: the coordinate $(6, 2)$ gets the value 15, because $(1 + x)^6 = 1 + 6x + 15x^2 + 20x^3 + 15x^4 + 6x^5 + x^6$ and thus we see that the coefficient of x^2 is 15. This gives us *the fourth version of Pascal's Triangle* (P$\triangle_4$).

Exercise 3.Y
Show that P$\triangle_4$, too, is equal to P$\triangle_1$.

We have seen that the four ways of presenting Pascal's Triangle all lead to the same triangle of numbers. Because we have seen that all versions coincide, we may speak of *the* Triangle of Pascal, without having to refer to any one of its specific instantiations.

Exercise 3.Z
Demonstrate how the triangle of Pascal can be used to figure out how many distinct ways there are, to pick four objects out of a collection of six. What is the notation for the corresponding binomial?

Theorem 3.48 (*Newton's Binomial Theorem*)
Let $x \in \mathbb{R}$ and $n \in \mathbb{N}$. Then

$$(1 + x)^n = \binom{n}{0} + \binom{n}{1}x + \binom{n}{2}x^2 + \cdots + \binom{n}{n-1}x^{n-1} + \binom{n}{n}x^n$$

Relating this theorem to what we have seen above, it actually simply is the statement 'P$\triangle_2$ is equal to P$\triangle_4$'.

Remark 3.49
There is also a more general version of Theorem 3.48 for $y, z \in \mathbb{R}$ and $x \in \mathbb{N}$:

$$(y+z)^n = \binom{n}{0}y^n + \binom{n}{1}y^{n-1}z + \binom{n}{2}y^{n-2}z^2 + \cdots + \binom{n}{n-1}yz^{n-1} + \binom{n}{n}z^n$$

It follows simply from the fact that if $y \neq 0$

$$\begin{array}{rl}
\multicolumn{2}{l}{(y+z)^n} \\
= & y^n\left(1+\left(\frac{z}{y}\right)\right)^n \\
= & y^n\left(\binom{n}{0}+\binom{n}{1}\left(\frac{z}{y}\right)+\binom{n}{2}\left(\frac{z}{y}\right)^2+\cdots+\binom{n}{n-1}\left(\frac{z}{y}\right)^{n-1}+\binom{n}{n}\left(\frac{z}{y}\right)^n\right) \\
= & \binom{n}{0}y^n+\binom{n}{1}y^n\left(\frac{z}{y}\right)+\binom{n}{2}y^n\left(\frac{z}{y}\right)^2+\cdots+\binom{n}{n-1}y^n\left(\frac{z}{y}\right)^{n-1}+\binom{n}{n}y^n\left(\frac{z}{y}\right)^n \\
= & \binom{n}{0}y^n+\binom{n}{1}y^{n-1}z+\binom{n}{2}y^{n-2}z^2+\cdots+\binom{n}{n-1}y^1z^{n-1}+\binom{n}{n}y^0z^n \\
= & \binom{n}{0}y^n+\binom{n}{1}y^{n-1}z+\binom{n}{2}y^{n-2}z^2+\cdots+\binom{n}{n-1}y^1z^{n-1}+\binom{n}{n}z^n
\end{array}$$

when $y \neq 0$. The case in which $y = 0$ is trivial as it comes down to

$$\begin{array}{rcl}
(0+z)^n & = & z^n \\
 & = & 0+0+0+\cdots+0+1\cdot zd^n \\
 & = & \binom{n}{0}0^n+\binom{n}{1}0^{n-1}z+\binom{n}{2}0^{n-2}z^2+\cdots+\binom{n}{n-1}0^1z^{n-1}+\binom{n}{n}z^n
\end{array}$$

We end this section about Pascal's triangles by adding up all elements on a row.

$$\begin{array}{ccccccccccccccccc}
 & & & & & & & 1 & & & & & & & & = & 1 \\
 & & & & & & 1 & + & 1 & & & & & & & = & 2 \\
 & & & & & 1 & + & 2 & + & 1 & & & & & & = & 4 \\
 & & & & 1 & + & 3 & + & 3 & + & 1 & & & & & = & 8 \\
 & & & 1 & + & 4 & + & 6 & + & 4 & + & 1 & & & & = & 16 \\
 & & 1 & + & 5 & + & 10 & + & 10 & + & 5 & + & 1 & & & = & 32 \\
 & 1 & + & 6 & + & 15 & + & 20 & + & 15 & + & 6 & + & 1 & & = & 64 \\
1 & + & 7 & + & 21 & + & 35 & + & 35 & + & 21 & + & 7 & + & 1 & = & 128
\end{array}$$

We hope you recognized the values 2^0, 2^1, 2^2, and so on.

3.10 Choosing k objects out of n objects

The binomial coefficient $\binom{n}{k}$ is pronounced as 'n choose k', hence it probably won't be a surprise that binomial coefficients play an important role in counting in how many ways one can choose k elements out of a set of n elements. However, this last sentence is not precise enough as there are two independent choices that play a role here:

- Are duplicates allowed or can we choose a specific element only once?
- Does the order in which the elements are chosen matter or not?

These two independent choices lead to four cases:

Theorem 3.50
The number of ways one can choose k elements out of n elements is given by this table:

	duplicates are not allowed	*duplicates are allowed*
the order does not matter	$\binom{n}{k}$	$\binom{n+k-1}{k}$
the order does matter	$\frac{n!}{(n-k)!}$	n^k

Let us explain the situations by looking at the ways to take two elements out of the set $\{1, 2, 3, 4, 5\}$.

- Duplicates are allowed and the order does matter. So the series $1, 2$ and $2, 1$ are not the same and series like $3, 3$ are allowed. How do we create such a series? For the first item we choose one out of five elements. For the second item we also choose one out of five elements. So we get 5^2 different series. In the general case, this extends to n^k series.

- Duplicates are not allowed and the order does matter. So the series $1, 2$ and $2, 1$ are not the same and series like $3, 3$ are not allowed. How do we create such a series? For the first item we can choose out of five elements, so there are five ways to pick the first element. For the second element, there are four options left as duplicates are not allowed. So we get $5 \cdot 4 = 20$ different series. For general n and k, the method of constructing a series works the same way: first we choose one out of n elements, then one out of $n-1$ elements, and so on. So there are $n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)$ ways to choose k elements. However, there is a shorter way to represent the same formula:

$$\begin{aligned}
&\frac{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)}{k!} \\
&= n \cdot (n-1) \cdot (n-2) \cdots (n-k+1) \cdot \frac{(n-k) \cdots 2 \cdot 1}{(n-k) \cdots 2 \cdot 1} \\
&= \frac{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1) \cdot (n-k) \cdots 2 \cdot 1}{(n-k) \cdots 2 \cdot 1} \\
&= \frac{n!}{(n-k)!}
\end{aligned}$$

- Duplicates are not allowed and the order does not matter. So the series $1, 2$ and $2, 1$ are now considered the same and series like $3, 3$ are still not allowed. How do we create such a series? Basically, in the same way as in the previous way, however, to compensate for the fact that the same series $1, 2$ and $2, 1$ are now actually counted twice, we divide by the number of ways that we can write down essentially the same series. That means, we have to divide by $2!$. So we get $\frac{5 \cdot 4}{2!} = \frac{20}{2} = 10$. And the general case is also the previous general case, but now divided by $k!$, so we get:

$$\frac{\frac{n!}{(n-k)!}}{k!} = \frac{n!}{n! \cdot (n-k)!} = \binom{n}{k}$$

- Duplicates are allowed and the order does not matter. So the series $1, 2$ and $2, 1$ are the same and series like $3, 3$ are allowed. How to create such a series? This is a difficult one! As the order does not matter, the essential information is how often each element appears in the series. So we should count how often we put a specific element in the series. But this is essentially a matter of putting balls in boxes! Let us explain this. We have five elements $1, 2, 3, 4, 5$ and for each element we create a box. For ease of use, we simplify the boxes a bit. This is what it looks like for the series $1, 2$ (and $2, 1$ as well!) and for the series $3, 3$:

$$\boxed{\circ \mid \circ \mid \quad \mid \quad \mid \quad} \qquad \boxed{\quad \mid \quad \mid \circ\circ \mid \quad \mid \quad}$$

The space in the empty boxes are not important for the characterization of the series, but the balls and the bars distinguishing the boxes are! So there are actually six important positions, and two of them are the balls and four of them are the bars. Hence in order to create a series, the only thing we have to do is picking the two positions of the two balls. (Or picking the four positions of the four bars!) However, this is a problem that we have seen before: it is an instance of picking two out of six elements (the possible positions for the balls), without duplicates and where the order is not important. So this can be done in $\binom{6}{2} = 15$ ways. The general case for picking k elements out of n elements leads to the formula $\binom{n+k-1}{k}$. The $n + k - 1$ is actually $(n - 1) + k$ as there are $n - 1$ bars and k balls.

3.11 Counting partitions

The binomial coefficients $\binom{n}{k}$ are used everywhere in mathematics. However, there are other famous families of numbers. An example of this are the *Stirling numbers*.

Actually there are two kinds of Stirling numbers: the Stirling numbers of the first kind or *Stirling cycle numbers* $\left[{n \atop k}\right]$, and the Stirling numbers of the second kind or *Stirling set numbers* $\left\{{n \atop k}\right\}$. The second kind are the more useful, and are the main subject of this section. When we write Stirling numbers without mentioning a kind, in these course notes, we mean the Stirling numbers of the second kind or Stirling set numbers.

Both kinds of Stirling numbers share many features with the binomial coefficients. For instance, for each kind there is also a triangle of numbers produced according to some recurrence relation. Both kinds have a combinatorial interpretation. The numbers of the first kind count permutations with a specific number of permutation cycles, but we will not discuss this combinatorial aspect any further in this course. The numbers of the second kind count partitions, which turns out to be the same as the number of ways to separate objects in separate piles. And for both kinds there is also a formula similar to the Binomial Theorem.

Now before we go to the formal recursive definitions and the triangles, let us start with counting arrangements of objects.

How many ways are there to put four objects into non-empty boxes? The objects will be distinct, let us call them $\{1, 2, 3, 4\}$, but the boxes will be indistinguishable.

- There are $\left\{{4 \atop 0}\right\} = 0$ ways to put the four objects in zero boxes.
- There is $\left\{{4 \atop 1}\right\} = 1$ way to put everything in one box:

  $$\{1, 2, 3, 4\}$$

- There are $\left\{{4 \atop 2}\right\} = 7$ ways to put the objects in two boxes:

  $$\begin{gathered}\{1, 2, 3\}, \{4\}\\ \{1, 2, 4\}, \{3\}\\ \{1, 3, 4\}, \{2\}\end{gathered}$$

$$\begin{gathered}\{1,2\},\{3,4\}\\ \{1,3\},\{2,4\}\\ \{1,4\},\{2,3\}\\ \{1\},\{2,3,4\}\end{gathered}$$

- There are $\genfrac\{\}{0pt}{}{4}{3} = 6$ ways to put the objects in three boxes:

$$\begin{gathered}\{1,2\},\{3\},\{4\}\\ \{1,3\},\{2\},\{4\}\\ \{1,4\},\{2\},\{3\}\\ \{1\},\{2,3\},\{4\}\\ \{1\},\{2,4\},\{3\}\\ \{1\},\{2\},\{3,4\}\end{gathered}$$

- There is $\genfrac\{\}{0pt}{}{4}{4} = 1$ way to distribute everything over four boxes:

$$\{1\},\{2\},\{3\},\{4\}$$

These numbers are on the fifth row of the following triangle:

$$\begin{array}{ccccccccccccccc}
&&&&&&&1&&&&&&& \\
&&&&&&0&&1&&&&&& \\
&&&&&0&&1&&1&&&&& \\
&&&&0&&1&&3&&1&&&& \\
&&&\boxed{0}&&\boxed{1}&&\boxed{7}&&\boxed{6}&&\boxed{1}&&& \\
&&0&&1&&15&&25&&10&&1&& \\
&0&&1&&31&&90&&65&&15&&1& \\
0&&1&&63&&301&&350&&140&&21&&1
\end{array}$$

Except for the left boundary with a single one at the top and zeros in all other rows, it looks pretty similar to Pascal's triangle. This left boundary is a bit weird and in many textbooks it is just omitted. However, as we will see later on when we give the recursive definition, it also has some benefits. And because this left boundary is not that interesting, it is sometimes referred to as the 'zeroth diagonal', just to be able to call the diagonal with all the ones the 'first diagonal', which happens to be practical as we will see.

This triangle can be computed in a way very similar to Pascal's triangle. The only difference is that the number on the right first has to be multiplied by the 'number of the diagonal' before being added to the number on the left. For example we have

$$301 = 31 + 3 \cdot 90$$

because the 90 is on the third diagonal (where the direction of the diagonal is from the bottom-left to the top-right). Similarly we have

$$350 = 90 + 4 \cdot 65$$

because the 65 is on the fourth diagonal.

This triangle is the triangle of the *Stirling numbers of the second kind*:

$$\begin{array}{c}
{0 \brace 0} \\
{1 \brace 0} \quad {1 \brace 1} \\
{2 \brace 0} \quad {2 \brace 1} \quad {2 \brace 2} \\
{3 \brace 0} \quad {3 \brace 1} \quad {3 \brace 2} \quad {3 \brace 3} \\
{4 \brace 0} \quad {4 \brace 1} \quad {4 \brace 2} \quad {4 \brace 3} \quad {4 \brace 4} \\
{5 \brace 0} \quad {5 \brace 1} \quad {5 \brace 2} \quad {5 \brace 3} \quad {5 \brace 4} \quad {5 \brace 5} \\
{6 \brace 0} \quad {6 \brace 1} \quad {6 \brace 2} \quad {6 \brace 3} \quad {6 \brace 4} \quad {6 \brace 5} \quad {6 \brace 6} \\
{7 \brace 0} \quad {7 \brace 1} \quad {7 \brace 2} \quad {7 \brace 3} \quad {7 \brace 4} \quad {7 \brace 5} \quad {7 \brace 6} \quad {7 \brace 7}
\end{array}$$

From this triangle you can read that there are ${7 \brace 3} = 301$ ways to put seven objects in three non-empty boxes. Recall that this also holds for the strange zeroth diagonal: there is exactly one way to put nothing in zero non-empty boxes and there are zero ways of putting something in zero non-empty boxes.

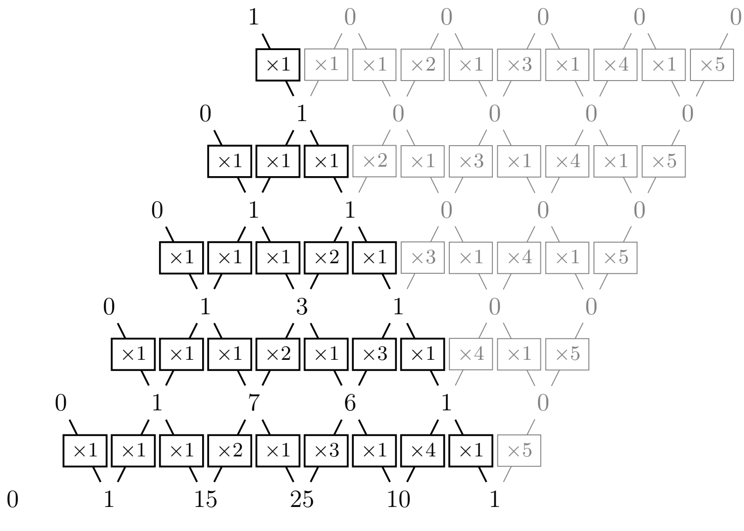
The values in this triangle can be computed with the following recursive definitions:

Definition 3.51
Let n and k be natural numbers. The *Stirling numbers of the second kind*, *Stirling set numbers*, ${n \brace k}$ are defined by these four equations:

$$\begin{array}{llll}
1. & \begin{Bmatrix}0\\0\end{Bmatrix} = 1 & \qquad 2. & \begin{Bmatrix}0\\k+1\end{Bmatrix} = 0 \\
3. & \begin{Bmatrix}n+1\\0\end{Bmatrix} = 0 & \qquad 4. & \begin{Bmatrix}n+1\\k+1\end{Bmatrix} = \begin{Bmatrix}n\\k\end{Bmatrix} + (k+1)\cdot\begin{Bmatrix}n\\k+1\end{Bmatrix}
\end{array}$$

Do you see the similarity with Definition 3.45 where we recursively defined the binomial coefficients?

We have already briefly explained how the values can be computed in the triangle, but now that we have given the formal recursive definition, we can look more clearly how the triangle is constructed.



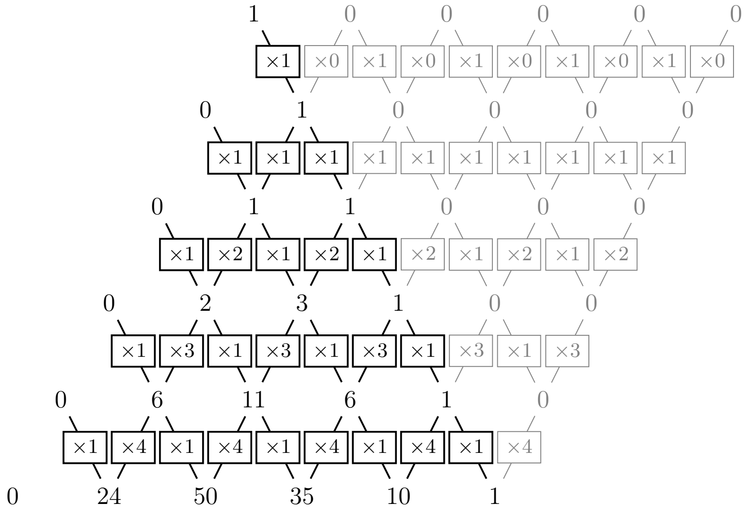
Values that are computed recursively have a line to two values on the row above, where the label refers to the multiplication factor. To the left, this factor is always one, but to the right it increases by one every next diagonal. As the values on the zeroth diagonal are taken directly from equations 1 or 3 from Definition 3.51, they don't have lines connecting them to other values. And on the right, you can see which 'gray' zeros coming from equations 2 and 4 are needed to construct the 'real' triangle.

Although we already indicated that the Stirling numbers of the second kind are the more interesting numbers, we do want to give you the recursive definition for the Stirling numbers of the first kind as well.

Definition 3.52
Let n and k be natural numbers. The *Stirling numbers of the first kind*, *Stirling cycle numbers*, $\begin{bmatrix} n \\ k \end{bmatrix}$ are defined by these four equations:

$$\begin{array}{llllll} 1. & \begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1 & \qquad 2. & \begin{bmatrix} 0 \\ k+1 \end{bmatrix} = 0 & & \\ 3. & \begin{bmatrix} n+1 \\ 0 \end{bmatrix} = 0 & \qquad 4. & \begin{bmatrix} n+1 \\ k+1 \end{bmatrix} = \begin{bmatrix} n \\ k \end{bmatrix} + n \cdot \begin{bmatrix} n \\ k+1 \end{bmatrix} & & \end{array}$$

The triangle for the Stirling cycle numbers is constructed in almost the same way as for the Stirling set numbers, however, this time the diagonal does not determine the multiplication factor, but the row.



Let us compare the recursive definitions of the three kinds of numbers that we are talking about here:

$$\begin{array}{ccccccc} \begin{pmatrix} n+1 \\ k+1 \end{pmatrix} & = & \begin{pmatrix} n \\ k \end{pmatrix} & + & & & \begin{pmatrix} n \\ k+1 \end{pmatrix} \\ \begin{bmatrix} n+1 \\ k+1 \end{bmatrix} & = & \begin{bmatrix} n \\ k \end{bmatrix} & + & n & \cdot & \begin{bmatrix} n \\ k+1 \end{bmatrix} \\ \begin{Bmatrix} n+1 \\ k+1 \end{Bmatrix} & = & \begin{Bmatrix} n \\ k \end{Bmatrix} & + & (k+1) & \cdot & \begin{Bmatrix} n \\ k+1 \end{Bmatrix} \end{array}$$

These are the relations for the binomial coefficients, and for the Stirling numbers of the first and second kind.

If we add *all* ways to separate four objects into boxes (regardless of the number of boxes), so we put all four kinds of possible splits that we showed before together, we get that there are

$$B_4 = 0 + 1 + 7 + 6 + 1 = 15$$

ways to do this. This motivates the following definition:

Definition 3.53
The *Bell numbers* are defined for $n \geq 0$ by:

$$B_n := \left\{ {n \atop 0} \right\} + \left\{ {n \atop 1} \right\} + \left\{ {n \atop 2} \right\} + \cdots + \left\{ {n \atop n} \right\}$$

Here are the first few Bell numbers:

$$\begin{array}{ccccccccccccccccc}
&&&&&&&1&&&&&&&&=&1\\
&&&&&&0&+&1&&&&&&&=&1\\
&&&&&0&+&1&+&1&&&&&&=&2\\
&&&&0&+&1&+&3&+&1&&&&&=&5\\
&&&0&+&1&+&7&+&6&+&1&&&&=&15\\
&&0&+&1&+&15&+&25&+&10&+&1&&&=&52\\
&0&+&1&+&31&+&90&+&65&+&15&+&1&&=&203\\
0&+&1&+&63&+&301&+&350&+&140&+&21&+&1&=&877
\end{array}$$

This is sequence A000110 in the *On-Line Encyclopedia of Integer Sequences* (OEIS), a huge database of sequences of numbers on the internet.

Definition 3.54
A *partition* of a set X is a set of nonempty subsets of X such that every element $x \in X$ is in *exactly one* of these subsets.

If we interpret each nonempty subset as a box, it is not difficult to see that all partitions of the set $\{1, 2, 3, 4\}$ are exactly the distributions over boxes that we have seen before, except that we put extra curly braces around them to make them a set:

$\{\{1,2,3,4\}\}$

$\{\{1,2,3\},\{4\}\}$
$\{\{1,2,4\},\{3\}\}$
$\{\{1,3,4\},\{2\}\}$
$\{\{1,2\},\{3,4\}\}$
$\{\{1,3\},\{2,4\}\}$
$\{\{1,4\},\{2,3\}\}$
$\{\{1\},\{2,3,4\}\}$

$\{\{1,2\},\{3\},\{4\}\}$
$\{\{1,3\},\{2\},\{4\}\}$
$\{\{1,4\},\{2\},\{3\}\}$
$\{\{1\},\{2,3\},\{4\}\}$
$\{\{1\},\{2,4\},\{3\}\}$
$\{\{1\},\{2\},\{3,4\}\}$

$\{\{1\},\{2\},\{3\},\{4\}\}$

Remark 3.55
As the official notation of these partitions requires a lot of curly braces, whereas the essential thing is actually to show which elements are grouped together, sometimes a different notation is used, where a '|' simply separates the groups of elements. So the partition $\{\{1,2,3,4\}\}$ becomes 1 2 3 4, $\{\{1,3,4\},\{2\}\}$ becomes 1 3 4 | 2, and $\{\{1\},\{2\},\{3,4\}\}$ becomes 1 | 2 | 3 4.

Remark 3.56
The Bell number B_n represents exactly the number of partitions of a set with n objects.

So there are $B_4 = 15$ partitions of the set $\{1, 2, 3, 4\}$ as can be checked above.

Exercise 3.AA
Assume we have one bill of each of the following types: \$1, \$2, \$5, \$10, \$20, \$50, and \$100. In how many ways can we divide all these bills over three identical purses if there is at most one empty purse?

Exercise 3.AB
Assume we have seven marbles. In how many ways can we distribute all these marbles over three bags (where bags may be empty) if

(i) both the marbles and the bags are distinguishable?
(ii) both the marbles and the bags are indistinguishable?
(iii) the marbles are indistinguishable and the bags are distinguishable? [*Hint:* What do you think that $\boxed{\circ\circ\circ \mid \circ\circ \mid \circ\circ}$ represents?]
(iv) the marbles are distinguishable and the bags are indistinguishable?
(v) the marbles are distinguishable and the bags are indistinguishable, but none of the bags may be empty?

Exercise 3.AC
(i) Explain by using a combinatorial argument that $\genfrac\{\}{0pt}{}{n}{2} = 2^{n-1} - 1$ for $n \geq 2$.
(ii) Prove by induction that $\genfrac\{\}{0pt}{}{n}{2} = 2^{n-1} - 1$ for $n \geq 2$.

Exercise 3.AD
Explain by using a combinatorial argument that $\genfrac\{\}{0pt}{}{n}{n-1} = \binom{n}{2}$ for $n \geq 2$.

Exercise 3.AE
Which of the following sets are partitions of the natural numbers? If it is not a partition, explain why not.
(i) $\{\mathbb{N}\}$
(ii) $\{\{37\}, \{42\}, \mathbb{N} \setminus \{37, 42\}\}$
(iii) $\{\{x \in \mathbb{N} \mid x \text{ is prime}\,(x)\}, \{x \in \mathbb{N} \mid x \text{ is not prime}\}\}$
(iv) $\{\{1\}, \{2\}, \{3\}, \ldots\}$
(v) $\{\{0, 1\}, \{1, 2\}, \{2, 3\}, \ldots\}$
(vi) $\{\{x \in \mathbb{N} \mid x \text{ is a multiple of } 2\}, \{x \in \mathbb{N} \mid x \text{ is a multiple of } 3\},$
$\{x \in \mathbb{N} \mid x \text{ is not a multiple of 2 and } x \text{ is not a multiple of } 3\}\}$

Exercise 3.AF
(i) The triangle of the Stirling numbers of the *first* kind has as its top:

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 0 & & 1 & & & & \\
 & & & 0 & & 1 & & 1 & & & \\
 & & 0 & & 2 & & 3 & & 1 & & \\
 & 0 & & 6 & & 11 & & 6 & & 1 & \\
0 & & 24 & & 50 & & 35 & & 10 & & 1
\end{array}$$

Calculate two more rows in this triangle.
(ii) Calculate the sum of the numbers in each row of the triangle you just calculated. Can you guess a formula for these sums?

Exercise 3.AG
Expand the polynomials

$$(1 + x)\,(1 + 2x) \cdots (1 + nx)$$

for $n \in \{0, 1, 2, 3, 4\}$. Do you see a relationship with the triangle from the previous exercise?

Exercise 3.AH
There are nice relations between the Stirling numbers of the first and second kind.
(i) Given numbers a_1, a_2, a_3, a_4, we define:

$$\begin{aligned}
b_1 &:= a_1 \\
b_2 &:= a_1 + a_2 \\
b_3 &:= 2a_1 + 3a_2 + a_3 \\
b_4 &:= 6a_1 + 11a_2 + 6a_3 + a_4
\end{aligned}$$

so with coefficients from the triangle of Stirling numbers of the first kind. Now from these numbers b_i we define:

$$\begin{aligned}
c_1 &:= b_1 \\
c_2 &:= -b_1 + b_2 \\
c_3 &:= b_1 - 3b_2 + b_3 \\
c_4 &:= -b_1 + 7b_2 - 6b_3 + b_4
\end{aligned}$$

with coefficients from the triangle of Stirling numbers of the second kind, but with the sign alternating between plus and minus.
Express the numbers c_i in terms of the a_i. What do you find?

(ii) Now we reverse the two kinds of Stirling numbers in this relation. If from the numbers c_i from the previous exercise we go on, and define:

$$\begin{aligned}
d_1 &:= c_1 \\
d_2 &:= c_1 + c_2 \\
d_3 &:= 2c_1 + 3c_2 + c_3 \\
d_4 &:= 6c_1 + 11c_2 + 6c_3 + c_4
\end{aligned}$$

and then express d_i in terms of b_i, what do we find then?

3.12 Important concepts

Chapter 4

Languages

Generating and describing languages is an important application of computers. In principle, a computer can only deal well with languages that are given an exact "formal" definition, for example, programming languages. Though one can of course learn a computer to recognize a natural language, as well, provided you give precise enough rules for it. But for now, we will deal with formally defined languages. We take a *language* to be a set of *words*. And a *word* is simply a sequence of symbols taken from a specified *alphabet*.

With these informal definitions we can already ask ourselves a number of interesting questions about languages and their words, like:

Question 4.1

- Does L contain the word w?
- Are the languages L and L' the same?

Many problems in computer science that seem on first sight unrelated to languages, can be reformulated, or, translated, into corresponding questions about (formal) languages.

Remark 4.2
As languages are sets, it is essential that you know how sets are represented in mathematical notation. So if you are not familiar with this, please read Appendix A.

4.1 Alphabet, word, language

Definition 4.3

1. An *alphabet* is a finite set Σ of *symbols*.[1]
2. A *word* over Σ is a finite number of symbols, strung together to a sequence.
3. λ is the word that is made up of 0 symbols, that is, no symbols at all, which is called the *empty word*.
4. Σ^* is the set of all words over Σ.
5. A *language* L over Σ is a subset of Σ^*.

Note that for *words* the order of the symbols matters, so $ab \neq ba$. In addition, duplication of symbols matters, so $a \neq aa$. And words cannot be infinite. Variables for words are u, v, w, $\ldots$.

[1] Although the alphabet may of course contain symbols such as ',' and '*', we will often call all symbols 'letters' for simplicity. And typically, a, b, c, ... are used as concrete symbols, whereas x, y, z, ... are variables used to indicate symbols.

However, for *languages* it is different. As languages are sets, the order of the words in the language doesn't matter. And also because languages are sets, duplication of words also doesn't matter. And languages can be infinite. Variables for languages are L, L', L_1, ...

Example 4.4

(i) $\Sigma = \{a, b\}$ is an alphabet.
(ii) *abba* is in Σ^* (notation: $abba \in \Sigma^*$).
(iii) $abracadabra \notin \Sigma^*$.
(iv) $abracadabra \in \Sigma_0^*$, with $\Sigma_0 = \{a, b, c, d, r\}$.
(v) The *empty word* λ is in Σ^*, whatever Σ may be.

We can describe languages in several ways. A couple of important ways, which we will treat in this course, are *regular expressions* (see Section 4.2), *grammars* (see Section 4.3), and *finite automata* (see Section 5.1). But remember that of course, languages are also simply the sets of words they contain.

Definition 4.5

We will introduce some useful notation.

1. We write a^n for $a \ldots a$ where we have n times a in a sequence. More precisely put: $a^0 = \lambda$, and $a^{n+1} = a^n a$.

2. The *length* of a word w is denoted $|w|$.

3. If w is a word, then w^R is the *reversed* version of the same word, so for example we have that $(abaabb)^R = bbaaba$.

Example 4.6

(i) $L_1 := \{w \in \{a, b\}^* \mid w \text{ contains an even number of } a\text{'s}\}$.
(ii) $L_2 := \{a^n b^n \mid n \in \mathbb{N}\}$.
(iii) $L_3 := \{wcv \in \{a, b, c\}^* \mid w$ does not contain any b, v does not contain any a, and $|w| = |v|\}$.
(iv) $L_4 := \{w \in \{a, b, c\}^* \mid w = w^R\}$.

Try, for every two of the above languages, to think of a word that is in the one, but not in the other.

If we have two languages L and L', we can define new languages with the usual set operations. (Though the complement might be new to you.)

Definition 4.7

Let L and L' be languages over the alphabet Σ.

1. Language L is a *subset* of language L' if each word in L is also a word in L', notation $L \subseteq L'$.

2. Language L is a *strict subset* of language L' if $L \subseteq L'$ and in addition there is at least one word in L' which is not in L, notation $L \subset L'$.

3. Language $\overline{L}$ is the *complement* of L: the language comprised of all words w that are *not* in L. (So: the $w \in \Sigma^*$ such that $w \notin L$.)

4. Language $L \cap L'$ is the *intersection* of L and L': the language comprised of all words w that are both in L as well as in L'.

5. Language $L \cup L'$ is the *union* of L and L': the language comprised of all words w that are either in L or in L' (or in both).

Exercise 4.A
In Example 4.6 we have seen some examples of language descriptions. Now try describing these languages yourself using formal set notation:
(i) Describe $L_1 \cap L_2$.
(ii) Describe $L_2 \cap L_4$.
(iii) Describe $L_3 \cap L_4$.

Besides these general operations on sets there are some operations that only work for languages, which we will now define:

Definition 4.8
Let L and L' be two languages of the alphabet Σ.

1. LL' is the *concatenation* of L and L': the language that contains all words of the shape wv, where $w \in L$ and $v \in L'$.

2. L^R is the language that contains all *reversed* words of L, that is, all w^R for $w \in L$.

3. L^* is the language comprised of all finite concatenations of words taken from L; so that it contains all words of the shape $w_1 w_2 \dots w_k$, where $k \geq 0$ and $w_1, w_2, \dots, w_k \in L$.

The language L^* is called the *Kleene closure* of L. The language L^* contains all concatenations of zero or more words of L, so it is always the case that $\lambda \in L^*$. Furthermore, we always have $L \subseteq L^*$ for any language L. Try to find out yourself what the language L^* is for $L = \varnothing$ and for $L = \{\lambda\}$.

Exercise 4.B
Use the definitions of Example 4.6.
(i) Prove that $L_1 = L_1{}^*$
(ii) Does $L_2 L_2 = L_2$ hold? Give a proof, or else a counterexample.
(iii) Does $\overline{L_1} = \overline{L_1}^*$ hold? Give a proof, or else a counterexample.
(iv) For which languages of Example 4.6 do we have $L = L^R$? (You need only answer, a proof is not necessary.)

4.2 Regular languages

A very popular way of describing languages is by the means of *regular expressions*. A language that can be described by such an expression, is called a *regular language*. In computer science, regular languages are seen as (relatively) simple languages: if L is such a language, it is not hard to build a little program that checks whether a given word w is contained in L. Such a program is then called a *parser*, and answering the question whether "$w \in L$" is called *parsing*. Parsers for regular languages are easy to make, and there are many programs that will even generate such parsers for you (when given as an input, a regular expression defining that language). The parsers generated by these "parser generators" are especially *efficient*: they decide very quickly whether w is or is not in L. This course will not further deal with the concept of parsing, but we will delve further into the notions of a regular language and a regular expression.

Definition 4.9
Let Σ be an alphabet. The *regular expressions* over Σ are defined as follows:

1. $\varnothing$ is a regular expression,

2. λ is a regular expression,

3. x is a regular expression, given any $x \in \Sigma$,

4. if r_1 and r_2 are regular expressions, then so too are the union $(r_1 \cup r_2)$ and the concatenation $(r_1\, r_2)$,

5. if r is a regular expression, then the Kleene star r^* is a regular expression too.

In order to be able to reduce the amount of parentheses, we provide the relative binding strength of the operators: Kleene star binds more strongly than concatenation, and concatenation binds more strongly than union. So $a \cup bc^*$ is the same expression as $(a \cup (b(c^*)))$, whereas it is $(a \cup (bc^*))$ according to the formal grammar above.

Example 4.10
Some examples of regular expressions are: aba, $ab^*(a \cup \lambda)$, $(a \cup b)^*$, $(a \cup \varnothing)b^*$, and $(ab^* \cup b^*a)^*$.

Remark 4.11
Let r and r' be regular expressions. And let n be a natural number. From Definition 4.9 it follows that the following series of symbols are *not* regular expressions: $\{r\}$, $\overline{r}$, $r \cap r'$, r^R, r^n, and r, r'.

Now let's formalize the relation between regular expressions and (regular) languages.

Definition 4.12
For every regular expression r, we define the *language of* r, denoted $\mathcal{L}(r)$, as follows:

1. $\mathcal{L}(\varnothing) := \varnothing$,

2. $\mathcal{L}(\lambda) := \{\lambda\}$,

3. $\mathcal{L}(x) := \{x\}$ for every $x \in \Sigma$,

4. $\mathcal{L}(r_1 \cup r_2) := \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$,

5. $\mathcal{L}(r_1\, r_2) := \mathcal{L}(r_1)\mathcal{L}(r_2)$,

6. $\mathcal{L}(r^*) := \mathcal{L}(r)^*$.

To exemplify this definition, take a look at the languages of the regular expressions from Example 4.10 above:

- $\mathcal{L}(aba) = \{aba\}$,

- $\mathcal{L}(ab^*(a \cup \lambda)) = \{a\}\{b\}^*\{a, \lambda\}$, so that is the language of words that start with an a, then an arbitrary amount of b's, and ending either with an a, or just ending at that,

- $\mathcal{L}((a \cup b)^*) = \{a, b\}^*$, or, all possible words over the alphabet $\{a, b\}$,

- $\mathcal{L}((a \cup \varnothing)b^*) = \{a\}\{b\}^*$, which is the same language as the one of the regular expression of ab^*,

- $\mathcal{L}((ab^* \cup b^*a)^*) = (\{a\}\{b\}^* \cup \{b\}^*\{a\})^*$. This language is somewhat less easily described in natural language.

Sometimes regular expressions are also written using the $+$ operator: the expression a^+ then stands for "1 or more times a." Technically speaking, this operator isn't needed, because instead of a^+, one can also simply write aa^*. (Check this.)

Exercise 4.C

(i) Demonstrate that the operator ?, for which $a^?$ stands for either 0 or 1 times a, doesn't have to be added to the regular expressions, as it can be defined using the existing operators.

(ii) What is $\mathcal{L}\left(\varnothing ab^*\right)$?

(iii) We define the language $L_5 := \{w \in \{a,b\}^* \mid w \text{ contains at least one } a\}$. Give a regular expression that describes this language.

(iv) Give a regular expression that describes the language L_1 from Example 4.6.

(v) Show that $\mathcal{L}\left(ab\left(ab\right)^*\right) = \mathcal{L}\left(a\left(ba\right)^* b\right)$.

We see that the regular expression $\varnothing$ isn't very useful. For any expression r, we can simply replace $r \cup \varnothing$ with r, so that the only reason to use $\varnothing$ is when we want to describe the empty language $\mathcal{L}(\varnothing)$. Apart from this use, you won't see the expression $\varnothing$ any more.[2]

Definition 4.13

Let Σ be an alphabet. We call a language L over Σ *regular* if some regular expression exists that describes it. More precisely put: a language L over Σ is regular if and only if there is a regular expression r for which $L = \mathcal{L}(r)$.

The language L_1 from Example 4.6 is regular, as we have seen in Exercise 4.C. However, the languages L_2, L_3, and L_4 from Example 4.6 are not. We won't be able to prove this with the material of this course, though.

Exercise 4.D

Show that the following languages are regular.

(i) $L_6 := \{w \in \{a,b\}^* \mid \text{every } a \text{ in } w \text{ is directly followed by a } b\}$,

(ii) $L_7 :=$ the language of all well-formed natural numbers. These words (numbers) are made up of the symbols $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$, but they never start with a 0, except for the word 0 of course.

(iii) $L_8 :=$ the language of all well-formed integers These words (integers) are made up of the natural numbers (for which you defined a regular expression in the previous question), possibly preceded by a $+$ or $-$ sign. [*Hint:* If you named the expression in the previous question, you may reuse it here.]

(iv) $L_9 :=$ the language of all well-formed arithmetical expressions without parentheses. These contain all natural numbers, possibly interspersed with the operators $+$, $-$, and $\times$, as in for example $7 + 3 \times 29 - 78$.

If a language L is regular, then the language L^R is also regular, because if L is described by some regular expression e (so $L = \mathcal{L}(e)$), then L^R is described by the regular expression e^R, because $L^R = \mathcal{L}(e^R)$. (Though, strictly speaking, we are not allowed to write e^R at all, as we have only defined the operation $\ldots^R$ on words and languages, and not on regular expressions. Try to figure out what a definition of *reversed* regular expressions would look like.)

Regular languages have more nice properties: if L and L' are regular, then so too are LL', $L \cup L'$, $\overline{L}$, and also $L \cap L'$. This is easily seen in the case of LL' and $L \cup L'$, but the other two are more complicated. Try to figure out yourself why $L \cap L'$ and $\overline{L}$ are regular too. Oh, and note that if we did not have that $\varnothing$ is a regular expression and hence $\mathcal{L}(\varnothing)$ not a regular language, then it would not be true that $\overline{r}$ is regular for all regular expressions. Can you think of a counterexample?

Convention 4.14

Sometimes, people identify a regular language with one of its describing regular expressions, and speak, say, of "the language $b^*(aab)^*$," although what is actually meant is the language

[2] Because $\mathcal{L}(\varnothing) = \varnothing$, we can succinctly use the same symbol to denote both the expression, as its language.

$\mathcal{L}(b^*(aab)^*)$. In this course, we try not to do this, but we try to make the difference between the regular expression and the corresponding language explicitly clear.

Exercise 4.E
Which of the following regular expressions describe the same language? For any two expressions, either show that they describe the same language, or else give a word that exemplifies that this is not the case.

(i) $b^*\,(aab)^*$.
(ii) $b^*\,(baa)^*\,b$.
(iii) $bb^*\,(aab)^*$.

Example 4.15
So far, we have used quite a lot of symbols in this chapter and some of these symbols can be used in different situations. Can you complete the following table?

	symbol	word	language	reg. exp
a	×	×		×
ab				
$\{a\}$				
a^*				
$\{a\}^*$				
λ				
$\varnothing$				
$a \cup b$				
$\{a, b\}$				

4.3 Context-free grammars

Let us now turn to another often used way of defining languages. Instead of attempting to *describe* a language directly, we give a method of giving a set of rules that describe how the words of the language are *generated* (or, *produced*). Such a set of rules is then called a "grammar."

Example 4.16
Let $\Sigma = \{a, b\}$. The language $L_{10} \subseteq \Sigma^*$ is defined by the *productions* starting with S using these *production rules*:

$$\begin{array}{rcl} S & \to & aAb \\ A & \to & aAb \\ A & \to & \lambda \end{array}$$

The method of generating words is then as follows. We start with S, called the *start symbol.* Both S and A are *nonterminals*. We follow an arrow of S, of which there is only one in this example (though there could have been more in general). If we still have nonterminals in our new word, we follow one of its arrows, and so on, until there are no nonterminals left, and we have successfully produced a word. Some example productions are:

$$\begin{array}{l} S \to aAb \to aaAbb \to aabb \\ S \to aAb \to aaAbb \to aaaAbbb \to aaabbb \end{array}$$

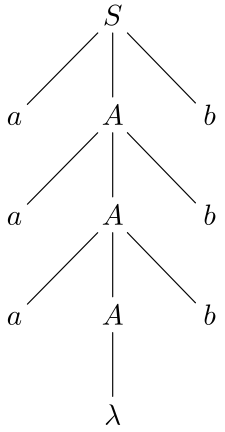
This way to represent a language is called a grammar. Let us give it the name G_1 so we can refer to it later on. The grammar above may also be written, more succinctly, as:

$$\begin{array}{rcl} S & \to & aAb \\ A & \to & aAb \mid \lambda \end{array}$$

By which we mean that, from A, two production rules are possible, namely $A \to aAb$ and $A \to \lambda$.
This grammar generates the language L_{10}, where $L_{10} = \{ab, aabb, aaabbb, a^4b^4, \ldots, a^nb^n, \ldots\}$. Or, written more clearly:

$$L_{10} = \{a^nb^n \mid n \in \mathbb{N} \text{ and } n > 0\}$$

Remark 4.17
We have already seen parse trees for propositional logic and for predicate logic. But we can also use parse trees to represent productions. We have seen above that *aaabbb* can be produced by the given grammar. This is the corresponding tree.



In the parse tree, a leaf corresponds to a word, and a non-leaf vertex corresponds to a nonterminal that produces the word parts depicted by the vertices drawn below it. So, following the production rules $S \to aAb$, we draw a vertex with label S, and three subvertices: a leaf with label a, a non-leaf with label A, and a leaf with label b. The next production step, $A \to aAb$, adds another level of subvertices with a leaf with label a, a non-leaf with label A, and a leaf with label b. This continues until all nonterminals are gone and we end up with the tree above. Reading the leaves, beginning at the top left, counter-clockwise around the graph, we get the produced word: *aaabbb*. Note that the empty word λ is not written when concatenated to non-empty words.

Definition 4.18
A *context-free grammar* G is a triple $\langle \Sigma, V, R \rangle$ consisting of:

1. An alphabet Σ.

2. A set of *nonterminals* V, such that $\Sigma \cap V = \varnothing$, and containing at least the special symbol S: the *start symbol.*

3. A set of *production rules* R of the form

$$X \to w$$

where X is a nonterminal and w a word made up of letters from the alphabet as well as nonterminals. (Put succinctly: $w \in (\Sigma \cup V)^*$.)

Convention 4.19
We will denote nonterminals by capital letters (S, A, B, etc), reserving lowercase for the alphabet letters (a, b, c, etc), which are also called terminals. Hence automatically $\Sigma \cap V = \varnothing$ as required by the definition.

Example 4.20
(i) The language L_{10} from Example 4.16 is produced by a context-free grammar.

(ii) The language L_{11} is generated by the context-free grammar $\langle \Sigma, V, R\rangle$ having $\Sigma = \{a\}$, $V = \{S\}$, and $R = \{S \to aaS, S \to a\}$. This grammar generates all words containing an odd number of a's.

(iii) The language L_{12} is generated by the context-free grammar $\langle \Sigma, V, R\rangle$ with $\Sigma = \{a, b\}$, $V = \{S, A, B\}$, and $R = \{S \to AB, A \to Aa, A \to \lambda, B \to Bb, B \to \lambda\}$. The language L_{12} consists of all words that start with a sequence of zero or more a's and is followed by a sequence of zero or more b's.

Definition 4.21
Languages generated by context-free grammars are called *context-free languages*. We denote the language generated by G as $\mathcal{L}(G)$.

Remark 4.22
Context-free languages are systematically studied in the course 'Languages and Automata' in the computing science curriculum.

Example 4.23
The language of well-formed arithmetical expressions, including parentheses, is context-free. (And *not* regular!) A possible grammar for this language is:

$$\begin{array}{lcl}
S & \to & L\,S\,O\,S\,R \mid G \\
L & \to & (\\
R & \to &) \\
O & \to & + \mid \times \mid - \\
G & \to & D\,C \\
D & \to & 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
C & \to & 0C \mid 1C \mid 2C \mid 3C \mid 4C \mid 5C \mid 6C \mid 7C \mid 8C \mid 9C \mid \lambda
\end{array}$$

Can you find productions for the expressions $(33 + (20 * 5))$ and $((33 + 20) * 5)$?

Exercise 4.F
The best way to show that a language is context-free is by giving a context-free grammar.

(i) Show that the language of *balanced parentheses expressions* is context-free. By this, we mean the expressions over $\{(,)\}$ where every opening parenthesis is closed with a parenthesis as well, so for example $(()(()))$ and $(())()$ are balanced, but $(()()))$ is not.

(ii) Show that the language L_1 from Example 4.6 is context-free.

(iii) Show that the language L_2 from Example 4.6 is context-free. (Check Example 4.16.)

(iv) Show that the language L_3 from Example 4.6 is context-free.

(v) Show that the language L_4 from Example 4.6 is context-free.

Exercise 4.G
Consider the grammar G_2:

$$\begin{array}{lcl}
S & \to & AS \mid Sb \mid \lambda \\
A & \to & aA \mid \lambda
\end{array}$$

(i) Write G_2 as a triple $\langle \Sigma, V, R\rangle$.

(ii) Give a production demonstrating that $aabb \in \mathcal{L}(G_2)$.

(iii) Can you give a production demonstrating that $bbaa \in \mathcal{L}(G_2)$ within three minutes?

Exercise 4.H
Consider the following grammar G_3 for the language L_{13}.

$$\begin{array}{lcl} S & \rightarrow & aSb \mid A \mid \lambda \\ A & \rightarrow & aAbb \mid abb \end{array}$$

The nonterminals are S and A, and $\Sigma = \{a, b\}$.
(i) Give productions of *abb* and *aabb*.
(ii) Which words does L_{13} contain?

With some exercise, it is not so hard to see which words are contained in L_{13}. However, it is not as easy to actually *prove* that it contains no other words. Usually, this is possible, depending on the complexity of the grammar, but for our current purposes, this is too hard. A simpler question is the following:

Question 4.24
The word *aab* is not contained in L_{13}. How does one go about showing this?

To show that a word *is* produced by a grammar, one only has to find a generating production, in which one then usually succeeds. But how to show that a grammar is *not* able of producing a word?

In computing science, the notion of an *invariant* was introduced to deal with this kind of proof.

Definition 4.25
An invariant of G is a property P that holds for all words that are generated by G.

To prove that P indeed holds for all $w \in \mathcal{L}(G)$ one needs to demonstrate that:
(i) P holds for S; and
(ii) that P is *invariant* under the production rules, meaning that, if P holds for some word v and v' can be produced from v, then P also holds for v'.

Remark 4.26
Note that invariants are defined in relation to grammars and not in relation to the corresponding languages. Hence stating something like P is an invariant of a language, makes no sense at all.

The definition of an invariant given above is a purely mathematical description. However, there exists also a graphical interpretation. Imagine that there is a rectangle that contains all words over $(\Sigma \cup V)^*$. For each of these words w, either $P(w)$ holds or $P(w)$ does not hold. So we can divide the rectangle into two parts: on the left we have the area where all the words that have the property reside and on the right the words reside that do not have the property, and these areas are divided by a border in the middle. And we can show production steps by drawing arrows between words v and v' exactly if there is a production $v \rightarrow v'$ according to the grammar. So some words will be connected to other words and some words will not be connected. If we rewrite the two properties for being an invariant in terms of this drawing, it comes down to checking that:
(i) The word S is in the left area.
(ii) If we take an arbitrary word v that is in the left area, and we apply one of the production steps, then the resulting word will also be on the left side. In other words, we have to check that there are no arrows crossing the border from the left to the right.

Note that it doesn't matter whether there are arrows crossing the border from the right to the left.

Example 4.27
Now let us draw such a diagram for our grammar G_1, for which we have seen in Example 4.16 that it produces the language

$$L_{10} = \{a^n b^n \mid n \in \mathbb{N} \text{ and } n > 0\}$$

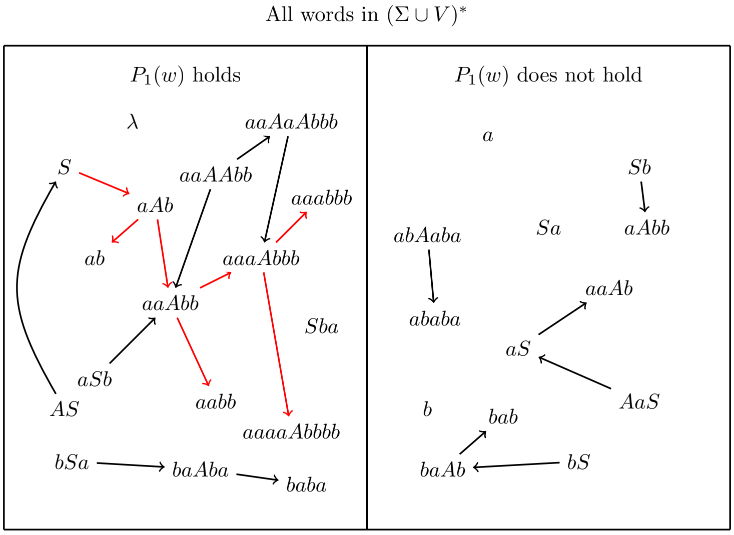
and two properties of words P_1 and P_2, where

$$P_1(w) := [\, w \text{ contains the same number of } a\text{'s and } b\text{'s.}]$$

and

$$P_2(w) := [\,\text{all } a\text{'s in } w \text{ occur before all } b\text{'s.}]$$

Let us place some random words in the rectangle and visualize some production steps by arrows in a diagram.



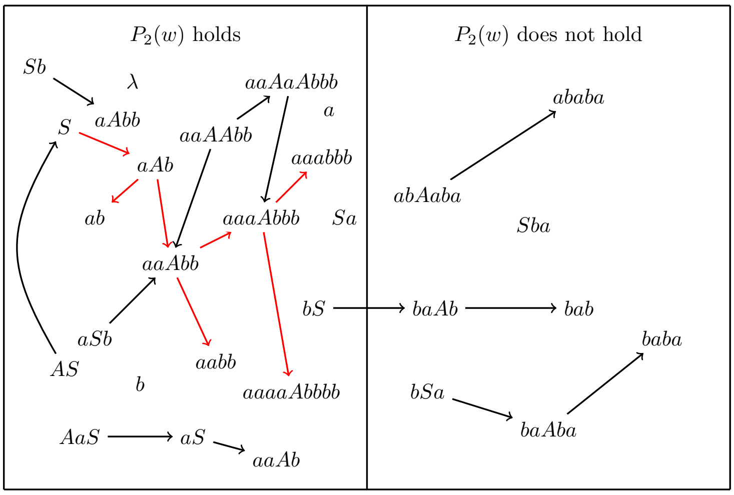
Some observations about this diagram:

- The diagram is not complete! Not all (infinitely many) words in $(\Sigma \cup V)^*$ are in the diagram.
- The diagram contains both words over $(\Sigma \cup V)^*$ that can be created with the grammar like $aaaAbbb$, but also words that cannot be created with the grammar like AS.
- All words over $(\Sigma \cup V)^*$ in this diagram that can be created by the grammar, can be found by following a series of red arrows starting in S.
- For some words, not all possible production steps are visualized by an outgoing arrow. This is only because the resulting word is not explicitly indicated in the diagram. For instance, on the left, $AS \to S$, $AS \to aAbS$, and $AS \to AaAb$ are valid productions, but only the first can be found, as the words $aAbS$ and $AaAb$ are not in the diagram.
- It is possible to talk about production steps from words that cannot be created with the grammar. See for instance the black production starting in bSa on the left.
- The word S is on the left.
- **There are no arrows crossing the border from left to right!**

The last two statements *suggest* that property P_1 is an invariant for this grammar. However, as we didn't put all possible words in the diagram, it could just be bad luck that we didn't put in that particular word on the left that has an outgoing arrow (black or red) to a word on the right. Therefore, this diagram method can *not* be used as a proof that some property is an invariant. Although P_1 is indeed an invariant for this grammar.

Now let us look at property P_2 and create a similar diagram with the same words as input. We get:

All words in $(\Sigma \cup V)^*$



Now note that:

- Some words have moved from the left to the right: Sba, bSa, $baAba$, and $baba$.
- Some words have moved from the right to the left: a, b, Sa, Sb, $aAbb$, AaS, aS, $aaAb$, and bS.
- The starting symbol S is on the left, so P_2 could still be an invariant for this grammar.
- **However, there is a black production $bS \to baAb$ that crosses the border from left to right!**

So even though it is not a red arrow crossing the border, but a black one, the conclusion is that P_2 is not an invariant.

Remark 4.28
Note that in the second property of an invariant, one must prove the invariance for *all* words v of $(\Sigma \cup V)^*$, not only the words that are contained in the grammar's language! In terms of the diagram: one should not only check that there are no red arrows crossing the border from left to right, but also that there are no black arrows. Given a word w and a property P, it is easy to check whether $P(w)$ holds or not, so it is easy to determine whether w should be to the left of the border or to the right. However, in general it is difficult to find out for a random word in $(\Sigma \cup V)^*$ whether it is reachable with red arrows or not. Therefore, the only way to be sure that a word that does not have property P is not reachable by red arrows from S is by knowing that there are no red arrows and no black arrows crossing the border.

To summarize, to prove that some word w is not in a grammar's language, using invariants, you do the following:

- Determine some "good" property P (called the invariant).
- Show that P holds for S.

- Show that P is invariant under the production rules.
- Show that w does not satisfy P.

Now let's get back to our problem.

Example 4.29
We want to show that $aab \notin L_{13}$, which is produced by grammar G_3. What would be a good invariant for the grammar G_3? We take

$$P(w) := [\text{the number of } b\text{'s in } w \geq \text{the number of } a\text{'s in } w]$$

This is indeed an invariant, because:

- $P(S)$ holds.
- If $P(v)$ holds and $v \longrightarrow v'$, then $P(v')$ holds as well. (Check this for every production rule: either both an a and a b are added, or an a and two b's, or neither an a nor a b; in all three cases, the number of b's stays greater or equal to the number of a's.)

  Note that, although Definition 4.25 talks of all words that are 'produced', it is only necessary to check all single step productions. (Try to convince yourself to find out why this is the case.)

So now we have proved that $P(w)$ holds for all words w produced by grammar G_3, so for all words $w \in L_{13}$. But because $P(aab)$ is obviously not true, we can then conclude that $aab \notin L_{13}$.

Exercise 4.I
We take another look at the grammar G_2 from Exercise 4.G. It was then already noted that $bbaa \notin \mathcal{L}(G_2)$. But can we prove that using an invariant? Let us give it a try with the predicate

$$P(w) := [\, w \text{ does not contain } ba \text{ as sub-word}]$$

At first this may seem like an invariant, but it isn't. Note that $P(bA)$ holds, because bA does not contain ba as sub-word. And note that applying the rule $A \to aA$ we get the production $bA \to baA$ and it is clear that $P(baA)$ does not hold. So P is not an invariant. But maybe we can try to fix this by putting more requirements in our predicate P...

(i) Is

$$P(w) := [\, w \text{ does neither contain } ba, \text{ nor } bA, \text{ nor } Sa \text{ as sub-word}]$$

an invariant for G_2 that proves that $bbaa \notin \mathcal{L}(G_2)$?

(ii) Is

$$P(w) := [\, w \text{ does neither contain } ba, \text{ nor } bA, \text{ nor } Sa, \text{ nor } bS \text{ as sub-word}]$$

an invariant for G_2 that proves that $bbaa \notin \mathcal{L}(G_2)$?

Exercise 4.J
Use invariants to prove that:
(i) $bba \notin L_{13}$.
(ii) $aabbb$ is not produced by the grammar of L_3 that you constructed in Exercise 4.F.
(iii) $aabbb$ is not produced by the grammar for L_4 that you constructed in Exercise 4.F.

Remark 4.30
Context-free grammars are called *context-free*, because the items on the left hand side of production rules are only allowed to be single nonterminals. So for example, the rule $Sa \to Sab$ is not allowed. And therefore, one never needs to take into account the *context* of the nonterminal (the symbols that may surround it in an intermediate step of production).

4.4 Right linear grammars

A well-known restricted form of context-free grammars are the *right linear grammars*.

Definition 4.31
A *right linear grammar* is a context-free grammar in which the production rules are always of the form

$$\begin{aligned} X &\rightarrow wY \\ X &\rightarrow w \end{aligned}$$

where X and Y are nonterminals and $w \in \Sigma^*$. Such a rule is called a *right linear rule*.

That is, in a right linear grammar, nonterminals are only allowed at the end of a rewrite, on the right hand side of a production rule.

Example 4.32
(i) In Example 4.20, only the grammar L_{11} is right linear.
(ii) Sometimes, for a context-free grammar that is not right linear, one can find an equivalent right linear grammar. For example, the following grammar (over $\Sigma = \{a, b\}$) is right linear and generates L_{12} from Example 4.20:

$$\begin{aligned} S &\rightarrow aS \mid B \\ B &\rightarrow bB \mid \lambda \end{aligned}$$

There is a mathematical theorem that states that the class of languages that can be produced by right linear grammars is exactly the class of regular languages.

Theorem 4.33
A language L is regular if and only if there is a right linear grammar that describes it.

Corollary 4.34
A regular language is always context-free.

We will not prove this theorem. To prove it, you have to show how to create a right linear grammar for every regular expression such that their languages are the same, and the other way around, creating a regular expression for each right linear grammar, such that their languages are the same. We will illustrate this by giving an example.

Example 4.35
Consider the regular expression

$$ab^*(ab \cup \lambda)(a \cup bb)^*$$

A right linear grammar producing the same language as that of the expression, is:

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow bA \mid B \\ B &\rightarrow abC \mid C \\ C &\rightarrow aC \mid bbC \mid \lambda \end{aligned}$$

Sometimes, right linear grammars are depicted with so-called *syntax diagrams*. We will just show an example of what such a diagram can look like. Figure 4.1 displays the syntax diagram corresponding to the grammar from Example 4.35.

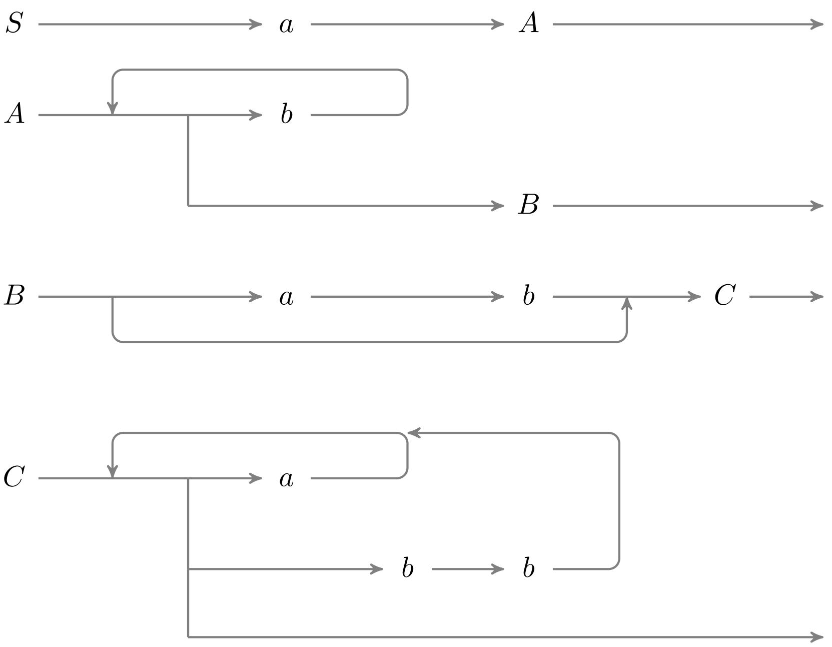


Figure 4.1: Syntax diagram for Example 4.35

Example 4.36
Recall the regular expression from Example 4.35:

$$ab^*(ab \cup \lambda)(a \cup bb)^*$$

A non-right linear grammar producing the same language as that of the expression, is:

$$\begin{array}{rcl} S & \to & aABC \\ A & \to & Ab \mid \lambda \\ B & \to & ab \mid \lambda \\ C & \to & aC \mid bbC \mid \lambda \end{array}$$

But how can that be? The language is regular and it is produced by a non-right linear grammar? Isn't that contradicting Theorem 4.33? No, it is not. The theorem only states that if the language is regular, then there exists at least one right linear grammar that produces the language and we have seen such a right linear grammar already in Example 4.35. The theorem does not state that all grammars that produce a regular language have to be right linear! In particular, being right linear (or not) is a property of a specific grammar and not of the language produced by this grammar.

Example 4.37
A direct result of Theorem 4.33 is that over the alphabet $\{a, b\}$, the following two languages are regular (see Example 4.20):

- $L_{11} = \{a^n \mid n \text{ is odd}\}$ and
- $L_{12} = \{wv \mid w \text{ contains only } a\text{'s and } v \text{ contains only } b\text{'s}\}$.

Try to construct regular expressions for these languages.

Example 4.38
Another consequence of Theorem 4.33 is that the language L_{14} of all words over $\{a, b\}$ that don't contain the word aa can be described in four ways:

- With natural language: as above.
- With a mathematical set notation: $\overline{\{uaav \mid u,v \in \{a,b\}^*\}}$.
- With a regular expression: $(b \cup ab)^*(a \cup \lambda)$.
- With a context-free right linear grammar:

$$S \quad \rightarrow \quad bS \mid abS \mid a \mid \lambda$$

Exercise 4.K

(i) Consider the following grammar over the alphabet $\{a,b,c\}$:

$$\begin{array}{rcl} S & \rightarrow & A \mid B \\ A & \rightarrow & abS \mid \lambda \\ B & \rightarrow & bcS \mid \lambda \end{array}$$

Check whether you can produce these words with the grammar: *abab*, *bcabbc*, *abba*. If you can, provide a production. If not, provide an invariant which you could use to prove that the word can not be produced.

(ii) Describe the regular language L_{15} that this grammar generates, with a regular expression.

(iii) Construct a right linear grammar for the language L_{16} consisting of all words of the shape $ab \ldots aba$ (that is, words with alternating a's and b's, starting and ending with an a; make sure to also include the word a).

Exercise 4.L

Give right linear grammars for the languages of Exercise 4.D:

(i) $L_6 := \{w \in \{a,b\}^* \mid \text{every } a \text{ in } w \text{ is directly followed by a } b\}$,

(ii) L_8 := the language of well-formed integer expressions. These consist of the symbols $0,1,2,3,4,5,6,7,8,9$, but never start with a 0, except for the word 0 itself, and may be preceded by a $+$ or a $-$ sign.

(iii) L_9 := the language of well-formed arithmetical expressions without parentheses. These consist of natural numbers, interspersed with the operators $+$, $-$, and $\times$, as in for example $7 + 3 \times 29 - 78$.

Exercise 4.M

Here we give a grammar for a small part of the English language.

$$\begin{array}{rcl} S = \langle\text{sentence}\rangle & \rightarrow & \langle\text{subjectpart}\rangle\langle\text{verbpart}\rangle\textbf{.} \\ \langle\text{sentence}\rangle & \rightarrow & \langle\text{subjectpart}\rangle\langle\text{verbpart}\rangle\langle\text{objectpart}\rangle\textbf{.} \\ \langle\text{subjectpart}\rangle & \rightarrow & \langle\text{name}\rangle \mid \langle\text{article}\rangle\langle\text{noun}\rangle \\ \langle\text{name}\rangle & \rightarrow & \textbf{John} \mid \textbf{Jill} \\ \langle\text{noun}\rangle & \rightarrow & \textbf{bicycle} \mid \textbf{mango} \\ \langle\text{article}\rangle & \rightarrow & \textbf{a} \mid \textbf{the} \\ \langle\text{verbpart}\rangle & \rightarrow & \langle\text{verb}\rangle \mid \langle\text{adverb}\rangle\langle\text{verb}\rangle \\ \langle\text{verb}\rangle & \rightarrow & \textbf{eats} \mid \textbf{rides} \\ \langle\text{adverb}\rangle & \rightarrow & \textbf{slowly} \mid \textbf{frequently} \\ \langle\text{adjectives}\rangle & \rightarrow & \langle\text{adjective}\rangle\langle\text{adjectives}\rangle \mid \lambda \\ \langle\text{adjective}\rangle & \rightarrow & \textbf{big} \mid \textbf{juicy} \mid \textbf{yellow} \\ \langle\text{objectpart}\rangle & \rightarrow & \langle\text{adjectives}\rangle\langle\text{name}\rangle \\ \langle\text{objectpart}\rangle & \rightarrow & \langle\text{article}\rangle\langle\text{adjectives}\rangle\langle\text{noun}\rangle \end{array}$$

(i) Is this grammar right linear?

(ii) Show how you produce the following sentence: *Jill frequently eats a big juicy yellow mango.*

(iii) Make some more sentences.

4.5 Grammar transformations

In this section we discuss three transformations. Each of these transformations, makes the grammar *cleaner* in some sense. This section is based on [3, Sections 4.2, 4.3, and 4.4], but adjusted to the definitions used in this course.

Example 4.39
In Example 4.16 we introduced language L_{10} produced by grammar G_1:

$$\begin{array}{rcl} S & \to & aAb \\ A & \to & aAb \mid \lambda \end{array}$$

We also listed two productions:

$$\begin{array}{l} S \to aAb \to aaAbb \to aabb \\ S \to aAb \to aaAbb \to aaaAbbb \to aaabbb \end{array}$$

Within these productions, we see that in the intermediate words over $(\Sigma \cup V)^*$ in these productions, there are some non-terminals A that do not produce any terminal symbols. This is due to the production step $A \to \lambda$. In a certain sense, this feels like a useless step. Therefore, the first transformation that we will discuss is about removing rules of the form $X \to \lambda$ for some non-terminal X. These rules are known as *λ-rules*.

In this situation it is not difficult to see that we can define grammar ${G_1}'$ by:

$$\begin{array}{rcl} S & \to & aAb \mid ab \\ A & \to & aAb \mid ab \end{array}$$

and we get that $\mathcal{L}({G_1}') = L_{10}$ and we have no λ-rule anymore. Our productions are indeed shorter now:

$$\begin{array}{l} S \to aAb \to aabb \\ S \to aAb \to aaAbb \to aaabbb \end{array}$$

However, this comes at the cost of having more rules in the grammar. Note that it is not sufficient to simply replace $A \to \lambda$ by $A \to ab$, as this would mean that there is no longer a production for the word ab, as $S \to aAb \to aabb$ would be the shortest production to a word.

Can we apply this transformation to all grammars to make sure that there are no λ-rules left anymore? No! There is an exception. If the language that is generated by a grammar contains the word λ, then somewhere there should be a way to produce this λ. So the best thing we can get is:

Theorem 4.40
Let $G = \langle \Sigma, V, R \rangle$ be a context-free grammar. Then there exists a grammar $G' = \langle \Sigma, V, R' \rangle$ such that:

- $\mathcal{L}(G) = \mathcal{L}(G')$,
- *if $\lambda \notin \mathcal{L}(G)$ then G' has no λ-rules at all, and*
- *if $\lambda \in \mathcal{L}(G)$ then G' has no λ-rules besides $S \to \lambda$.*

We will not give a proof, but we do give a construction of R'. In order to do this, we first introduce the concept of nullable non-terminals.

Definition 4.41
A non-terminal X is called *nullable* if it can produce the empty word λ. In other words, a non-terminal X is nullable if there exists a production $X \to w_1 \to w_2 \to \ldots w_n \to \lambda$ where each word $w_i \in V^*$.

Note that only non-terminals are allowed in the words w_i, because as soon as one of them contains a terminal, this terminal cannot be removed anymore, so the result can never be the empty word.

Example 4.42
From the definition it follows that being nullable is not just a matter of going in a single step from X to λ. For instance, let us consider grammar G_4:

$$\begin{array}{rcl} S & \to & ACC \mid bB \\ A & \to & aA \mid bB \mid C \\ B & \to & Bb \mid b \\ C & \to & Cc \mid \lambda \end{array}$$

Then C, A, and S are nullable, since we have the following productions:

$$\begin{array}{l} C \to \lambda \\ A \to C \to \lambda \\ S \to ACC \to CCC \to CC \to C \to \lambda \end{array}$$

The reason why we wrote the unusual order C, A, and S, is because this is the order in which you can find them: first look for the non-terminals that are nullable in one step, then for the non-terminals that are nullable in two steps, and so on. So it is a recursive procedure to find the nullable non-terminals.

Now that we know what nullable non-terminals are, we can describe our change from R to R'. We start by taking $R' := R$. Then we find all nullable non-terminals. And then, for each rule in R of the form $X \to w$ where

$$w = w_1X_1w_2X_2w_3 \dots w_nX_nw_{n+1}$$

where all non-terminals X_i are nullable, we add a new rule

$$X \to w_1w_2w_3 \dots w_nw_{n+1}$$

to R'. Once that is done for all rules and all possible combinations of w_i and X_i, we remove the λ-rules from R', except for $S \to \lambda$, if it is there.

Remark 4.43
The 'all possible combinations of w_i and X_i' may seem weird. But the problem is that it is not good enough to simply remove all nullable non-terminals in one go. Let us consider the rule $A \to aBAB$ where B is nullable. If we create a table for the w_i (where we allow $w_i = \lambda$ only for w_1 and w_{n+1}) and X_i we get:

w_1	X_1	w_2	X_2	w_3	new rule
a	B	A	B		$A \to aA$
a	B	AB			$A \to aAB$
aBA	B				$A \to aBA$

In other words, it is allowed to have nullable non-terminals inside w_i. Hence, instead of just adding $A \to aA$ we also need to add $A \to aAB$ and $A \to aBA$.

Example 4.44
If we apply this strategy on G_1 in Example 4.39, we see that A is nullable, but S is not. There are two rules that have nullable non-terminals on the right: $S \to aAb$ and $A \to aAb$. So we need to add both $S \to ab$ and $A \to ab$ to R'. After adding these rules, the only λ-rule that we have in R' is $A \to \lambda$. And after removing this we end up with ${G_1}'$ that we already showed in Example 4.39.

Exercise 4.N
Consider the grammar G_5:

$$\begin{aligned} S &\to AB \mid BCS \\ A &\to aA \mid C \\ B &\to bbB \mid b \\ C &\to cC \mid \lambda \end{aligned}$$

(i) Give all the words of $\mathcal{L}(G_5)$ that have a length not exceeding three.
(ii) What are the nullable non-terminals in G_5?
(iii) Provide an equivalent grammar ${G_5}'$ that has no λ-rules.

The second transformation that we are going to discuss is the elimination of so-called chain rules.

Definition 4.45
A rule of the form $X \to Y$ where both X and Y are non-terminals is called a *chain rule*. And a production

$$X_1 \to X_2 \to \cdots \to X_n$$

where each rule $X_i \to X_{i+1}$ is a chain rule, is called a *chain*. And the set *chain of* X is defined as the set of all non-terminals that are reachable via a chain of length at least zero.

These rules look 'suspicious' as they don't add anything, but only 'rename' a non-terminal. So the idea is that we are going to replace a chain rule $X \to Y$ by all rules $X \to w$ for all rules $Y \to w$. So we are substituting the result of Y directly into the rule for X.

However, the naive way of doing this, may introduce a new chain rule $X \to Z$ if there was a chain rule $Y \to Z$. So we are going to use the concept of chains, to prevent this.

Again, there is a theorem that explains that we can reduce these chain rules and end up with an equivalent grammar.

Theorem 4.46
Let $G = \langle \Sigma, V, R \rangle$ be a context-free grammar, without λ-rules, except for $S \to \lambda$. Then there exists a grammar $G' = \langle \Sigma, V, R' \rangle$ such that:

- *$\mathcal{L}(G) = \mathcal{L}(G')$ and*
- *G' has no chain rules.*

As usual, we will not give a proof, but we do give a construction of R'.

So how does it work? We first determine all chains of all non-terminals. Then, for all non-terminals X we add a rule $X \to w$ if there is a non-terminal Y and a string w such that:

- Y is in the chain of X,
- the rule $Y \to w$ is in R, and
- $w \notin V$.

And after that, we remove the chain rules.

Example 4.47
Let us consider the grammar G_4 again from Example 4.42:

$$\begin{aligned} S &\to ACC \mid bB \\ A &\to aA \mid bB \mid C \\ B &\to Bb \mid b \\ C &\to Cc \mid \lambda \end{aligned}$$

This grammar has a λ-rule that is not $S \to \lambda$, so we first have to eliminate the λ-rules. If we do that, we end up with grammar ${G_4}'$:

$$\begin{array}{rcl} S & \to & ACC \mid bB \mid CC \mid AC \mid C \mid A \mid \lambda \\ A & \to & aA \mid bB \mid C \\ B & \to & Bb \mid b \\ C & \to & Cc \mid c \end{array}$$

Now if we compute the chains, we get that the chain of $S = \{S, A, C\}$, the chain of $A = \{A, C\}$, the chain of $B = \{B\}$, and the chain of $C = \{C\}$. If we follow the algorithm, it is clear that for B and C we don't have to add anything, as their chains are trivial. For A we have to add the rules $A \to Cc$ and $A \to c$, as these can be derived in one step from C. For S we have to add the rules $S \to aA$ and $S \to bB$ as these can be derived in one step from A, and we have to add the rules $S \to Cc$ and $S \to c$, as these can be derived in one step from C. And we remove the rules $S \to C$, $S \to A$, and $A \to C$. So we end up with grammar ${G_4}''$:

$$\begin{array}{rcl} S & \to & ACC \mid bB \mid CC \mid AC \mid Cc \mid c \mid aA \mid bB \mid \lambda \\ A & \to & aA \mid bB \mid Cc \mid c \\ B & \to & Bb \mid b \\ C & \to & Cc \mid c \end{array}$$

Exercise 4.O
Consider the grammar G_6:

$$\begin{array}{rcl} S & \to & AS \mid A \\ A & \to & aA \mid bB \mid C \\ B & \to & bB \mid b \\ C & \to & cC \mid B \end{array}$$

(i) Give productions for the words *b*, *ab*, *bb*, *acb*, *bab*, *cab*, or explain why such a production doesn't exist.
(ii) Provide an equivalent grammar ${G_6}'$ that has no chain rules.
(iii) Check for the words in (i) that they have a production with grammar G_6 if and only if they have a production with grammar ${G_6}'$.

The last transformation that we will discuss is the elimination of useless symbols. So we start by defining which symbols are useful and/or useless.

Definition 4.48
Let $G = \langle \Sigma, V, R \rangle$ be a context-free grammar. A symbol $x \in \Sigma \cup V$ is *useful* if there is a production

$$S \to \cdots \to uxv \to \cdots \to w$$

where $u, v \in (\Sigma \cup V)^*$ and $w \in \Sigma^*$. A symbol x is *useless* if it is not useful. And a word $w \in \Sigma^*$ is called a *terminal string*.

As with the previous reductions there is a theorem that states that it is possible to eliminate the useless symbols from Σ and V.

Theorem 4.49
Let $G = \langle \Sigma, V, R \rangle$ be a context-free grammar. Then there exists a grammar $G' = \langle \Sigma', V', R' \rangle$ such that:

- *$\mathcal{L}(G) = \mathcal{L}(G')$ and*

- *G' has no useless symbols.*

As usual, we will not give a proof, but we do give a construction of Σ', V', and R'.

The first step is to determine the non-terminals that lead to terminal strings. This coincides with the

$$uxv \to \cdots \to w$$

part of the definition of useful symbols. We call these non-terminals *potentially useful.* This is done in a similar way as in the algorithm that finds the nullable variables. We start with listing the non-terminals that go to a terminal string in one step, as these are clearly potentially useful. Next, we determine the non-terminals that go to strings containing only symbols in Σ and/or non-terminals that are already known to be potentially useful. This process is repeated until no new non-terminals are added. Non-terminals that are not potentially useful, are certainly not useful symbols and can be removed from V and R in the grammar.

The next step is to see which of these potentially useful non-terminals can actually be reached from S, so it deals with the

$$S \to \cdots \to uxv$$

part of the definition of useful symbols. This is done in a similar way that we found the chains of S. We start with S and add all non-terminals that appear in the right hand sides of S. In the next round, we add all non-terminals that appear in the right hand sides of these newly added non-terminal. We continue until there is nothing more to add. Non-terminals that are not reachable from S will not add anything to the language, so they are useless and can be removed from V and R.

So far, we only looked at the reduction of V and R. However, if the alphabet Σ contains symbols that are never used in R, then we can also remove these from Σ. Now let us see how this works in practice.

Example 4.50
Consider the context-free grammar $G_7 = \langle\{a, b, c, d\}, \{S, A, B, C, D\}, R\rangle$ where R is given by:

$$\begin{array}{rcl} S & \to & AA \mid CD \mid bD \\ A & \to & aA \mid a \\ B & \to & bB \mid bC \\ C & \to & cB \\ D & \to & dD \mid d \end{array}$$

We start by finding the potentially useful non-terminals. The non-terminals A and D clearly lead to terminal strings a and d respectively in one step, so A and D are potentially useful. As $S \to AA$, S leads to a string that only consists of potentially useful non-terminals, so it is potentially useful itself. The non-terminals B and C do not lead to strings that only contain A's, D's, or elements of Σ, so these are not potentially useful yet. In the next round, we only have to check whether the remaining non-potentially useful non-terminals, can lead to strings that only contain A's, D's, S's, or elements of Σ, but that is also not the case. So in this round we didn't add any new potentially useful non-terminals. Hence B and C can be removed from V and R. This leads to the following grammar $G_7{}' = \langle\{a, b, c, d\}, V', R'\rangle$ where $V' = \{S, A, D\}$ and where R' is:

$$\begin{array}{rcl} S & \to & AA \mid bD \\ A & \to & aA \mid a \\ D & \to & dD \mid d \end{array}$$

We now have to check whether all non-terminals are reachable from S and that is the case. So we don't have to remove anything from V' and R' anymore, but we do need to check for possible useless symbols in $\{a, b, c, d\}$. As only a, b, and d occur in R', we get that the grammar $G_7'' = \langle\{a, b, d\}, V', R'\rangle$ is a grammar that produces the same language as G_7, but has no useless symbols.

Exercise 4.P
Consider the grammar $G_8 = \langle\{a, b\}, \{S, A, B, C, D, E, F, G\}, R\rangle$ where R is given by:

$$\begin{array}{rcl}
S & \rightarrow & aA \mid BD \\
A & \rightarrow & aA \mid aAB \mid aD \\
B & \rightarrow & aB \mid aC \mid BF \\
C & \rightarrow & Bb \mid aAC \mid E \\
D & \rightarrow & bD \mid bC \mid b \\
E & \rightarrow & aB \mid bC \\
F & \rightarrow & aF \mid aG \mid a \\
G & \rightarrow & a \mid b
\end{array}$$

(i) Give all six words of $\mathcal{L}(G_8)$ that have a length not exceeding five.
(ii) List the potentially useful non-terminals of G_8.
(iii) Which of the potentially useful non-terminals are not reachable from S?
(iv) Provide an equivalent grammar $G_8{}'$ that has no useless symbols. Make sure to write the full triple!
(v) Can you now prove that $\mathcal{L}(G_8)$ is a regular language?

To end this chapter about languages, let's get back to a question asked earlier:

Example 4.51
Did you manage to fill out the table in Example 4.15? Here is the solution:

	symbol	word	language	reg. exp
a	×	×		×
ab		×		×
$\{a\}$			×	
a^*				×
$\{a\}^*$			×	
λ		×		×
$\varnothing$			×	×
$a \cup b$				×
$\{a, b\}$			×	

4.6 Important concepts

Chapter 5

Automata

Recall the main question we discussed in Chapter 4:

Question 5.1

- Does L contain the word w?
- Are the languages L and L' the same?

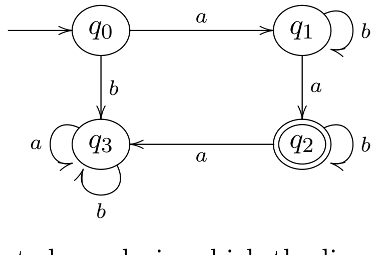
From the topic of languages, it is just a small step to the topic of *automata.* Automata are yet again formal mathematical objects, and the key question that we often ask about them, is which language they accept (or, define). This question relates the two, and is the reason why we treat automata directly after languages.

5.1 Automata

At the start of this chapter, we stated that languages can also be formalized using automata. To explain this connection, we first define what an automaton is.

In computer science, there is the study of machines. For example by looking at computers themselves, but also at a higher, conceptual level, computer scientists study idealized, abstract machines. An advantage of studying such an idealized and abstract *machine model* is that it is easier to study the important properties they have. A prolific class of these abstract machines are the *finite automata.* These finite automata have many more applications than just modeling simple calculations (as machines usually do). Finite automata, and simple extensions, can model processes as well, for example. Let us first take a look at what a finite automaton is, and how one 'calculates' with it. Here is an example:

M_1 :



(5.1)

An automaton is a so-called directed graph, in which the lines are *arrows* and have *labels*. (See Section 3.1 for a formal account of graphs.) The nodes of the graph are referred to as the *states* of the automaton, here: q_0, q_1, q_2, and q_3. The states are commonly drawn as circles with their name written inside. There are two distinguished types of states that have a special role:

1. The *initial state*, distinguished by an incoming arrow that doesn't depart from any other state. Every automaton has *exactly one* initial state, often named q_0 as in the example above.

2. The *final states*, distinguished by drawing their circle with a double border. The example above has exactly one, namely q_2. (It is allowed to have the initial state simultaneously be a final state.)

We can regard an automaton as a (very) simple computer that can compute things for us. This computation happens in the following way: you give it a word (in the example, this would be a word over the alphabet $\{a, b\}$), then the automaton computes according to the word, following the appropriate arrow for each subsequent letter of the given word, and then after a number of steps *halts*, either in a final state, or in a non-final state. In the first case, we say the automaton *accepts* the word, in the second, it *rejects* the word.

Example 5.2
The automaton M_1 accepts the word *aa*: it starts in state q_0, and then reads the first letter a. This brings it to the state q_1, and it then reads the second letter, another a. This then brings it to state q_2. There is no input any more, and the automaton has halted. Because it halted in a final state, *aa* is accepted by the automaton. We depict this *computation* with the notation:

$$q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_2$$

So, the notation shows how each subsequent letter is consumed, but does not depict whether the word has been accepted or not, which means you have to add this conclusion in writing. In this case, the computation ends in the final state q_2, and thus *aa* is accepted. Check that the words *abbba* and *ababb* are accepted as well, though *ababab* and *bab* are not.

Before continuing, we give a formal definition of the notion of a deterministic finite automaton.

Definition 5.3
A *deterministic finite automaton* is a quintuple $M := \langle \Sigma, Q, q_0, F, \delta \rangle$:

1. A finite set Σ, the input alphabet, or set of *atomic actions*,
2. A finite set Q of *states*,
3. A distinguished state $q_0 \in Q$ called the *initial state*,
4. A set of distinguished states $F \subseteq Q$, the *final states*,
5. A *transition function* δ, that maps every tuple of a state q and an action a to a state q'. (These are the labeled arrows in (5.1).)

A deterministic finite automaton is also called a *DFA*. Later on we will also introduce non-deterministic finite automata.

Note that the name 'final state' is a bit misleading, as it sort of suggests that the automaton stops in such a state. But it doesn't stop in these states if there is more input available! In that case it will simply apply the transition function δ on the next input and move to a (possibly different) state, which doesn't have to be a final state. Hence the name '*accepting state*' probably describes more clearly what is going on, but as these states are everywhere in literature called final states, we also use that terminology.

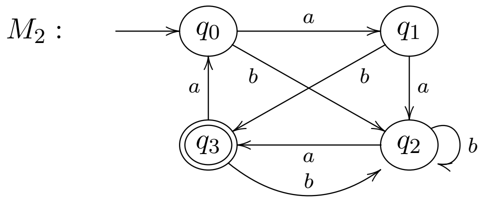
Example 5.4
This means that the automaton M_1 depicted in (5.1) is actually the quintuple $\langle \Sigma, Q, q_0, F, \delta \rangle$, where $\Sigma = \{a, b\}$, $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_2\}$, and the transition function δ is defined by:

$$\begin{array}{rclcrcl}
\delta(q_0, a) & = & q_1 & \qquad & \delta(q_0, b) & = & q_3 \\
\delta(q_1, a) & = & q_2 & & \delta(q_1, b) & = & q_1 \\
\delta(q_2, a) & = & q_3 & & \delta(q_2, b) & = & q_2 \\
\delta(q_3, a) & = & q_3 & & \delta(q_3, b) & = & q_3
\end{array}$$

Specifically, note how the deterministic behavior is captured by the fact that for every combination of state and symbol of the alphabet, exactly one transition is defined.

Instead of writing such a mathematical definition out in full, we usually simply draw a diagram as in the example on page 85.

Exercise 5.A
Consider the deterministic finite automaton M_2.



Here, $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_3\}$, and $\Sigma = \{a, b\}$.

(i) Check whether these words are accepted or not: *abaab*, *aaaba*, *bab*, λ, and *aabbab*.

(ii) Are these statements true? Give a proof or counterexample.

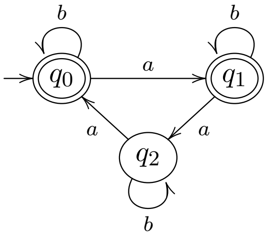
(1) If w is accepted, then so is $wabba$.

(2) If w is accepted, then wab is not accepted.

(3) If w is not accepted, then waa will not be accepted either.

(4) If w is not accepted, then neither is wbb.

Exercise 5.B
Give all words of length three that are accepted by the following deterministic finite automaton M_3:



Explain why these words are accepted.

5.2 Languages and automata

We can think of Σ as the set of "atomic actions" that lead us from one state to another, but also as the set of symbols of an alphabet, as we have done already above. If we think of Σ as representing the alphabet, the automaton can be seen as a *language recognizer*. In this way, for each automaton, there is a corresponding language, namely the language recognized by the automaton.

Definition 5.5
For a deterministic finite automaton $M := \langle \Sigma, Q, q_0, F, \delta \rangle$, we define the *language of* M, to be $\mathcal{L}(M)$:

$$\mathcal{L}(M) := \{w \in \Sigma^* \mid w \text{ is accepted by } M\}.$$

So: $w \in \mathcal{L}(M)$ if and only if the automaton M halts in a final state after consuming all of w.

Let us take a look at our initial automaton M_1, from page 85. It accepts the following words:

- *aa* is accepted,

- awa is accepted, with w an arbitrarily long sequence of b's,
- $awav$ is accepted, with w and v both arbitrarily long sequences of b's.

If we go "past" state q_2, we can never get back to a final state, and so the above description lists all words that the automaton accepts. Summarized:

$$\mathcal{L}(M_1) = \{ab^n ab^m \mid n, m \geq 0\}$$

Because we learned regular languages and expressions previously, we see that this language indeed is described by a regular expression, namely:

$$\mathcal{L}(M_1) = \mathcal{L}(ab^*ab^*).$$

Trying to find a similar corresponding regular expression for M_2, turns out to be a bit harder:

- ab is accepted,
- ba is accepted,
- aaa is accepted,
- $aab^k a$ is accepted, with $k \geq 0$,
- $bb^k a$ is accepted, with $k \geq 0$,
- $aab^k a(ba)^l$ is accepted, with $k \geq 0$, $l \geq 0$,
- $bb^k a(ba)^l$ is accepted, with $k \geq 0$, $l \geq 0$,

…and this is not all, because if we go "past" state q_2, we can in fact loop back to q_2 again. How can we then systematically analyze the language of an automaton? A method for doing so is constructing a corresponding *grammar*, that *generates* the same language as the automaton recognizes. This is done in this way:

1. For every state q_i, introduce a nonterminal X_i, and distinguish the starting nonterminal S for the initial state q_0.
2. For every transition $q_i \xrightarrow{a} q_j$ in the automaton, add the production rule $X_i \to aX_j$.
3. For every final state $q_i \in F$, add the production rule $X_i \to \lambda$.

Constructing the grammar G_9 (with $\Sigma = \{a, b\}$) for the automaton M_2, we then get:

$$\begin{array}{rcl} S & \to & bB \mid aA \\ A & \to & aB \mid bC \\ B & \to & bB \mid aC \\ C & \to & bB \mid aS \mid \lambda \end{array}$$

Note that this grammar is *right linear*, and so the language $\mathcal{L}(M_2)$ is indeed *regular*.

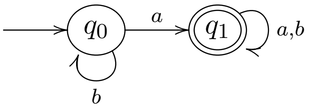
This new description of the language $\mathcal{L}(M_2)$, using a grammar, is interesting, if only because it is a new description that we haven't seen before. But what happens if we try to 'optimize' this grammar by substituting symbols? First, inline the rule $A \to aB \mid bC$ into the rule $S \to bB \mid aA$ to get $S \to bB \mid aaB \mid abC$. Then, do the same for the C rule. This gives us a new grammar yet again, G_{10}, still generating the same language, and still right linear as well:

$$\begin{array}{rcl} S & \to & bB \mid aaB \mid abbB \mid abaS \mid ab \\ B & \to & bB \mid abB \mid aaS \mid a \end{array}$$

Exercise 5.C
Conclude from the grammar G_{10} that

(i) $(aba)^k ab \in \mathcal{L}(G_{10})$ for all $k \geq 0$,
(ii) $aab^k a \in \mathcal{L}(G_{10})$ for all $k \geq 0$,
(iii) if $w \in \mathcal{L}(G_{10})$, then also $abaw \in \mathcal{L}(G_{10})$,
(iv) if $w \in \mathcal{L}(G_{10})$, then also $aaaaw \in \mathcal{L}(G_{10})$,

Exercise 5.D
Consider the deterministic finite automaton M_4:



Which of the following regular expressions does *not* describe the language of this automaton? Choose one of the options and provide an explanation.

(i) $b^*a\,(a \cup b)^*$
(ii) $(a \cup b)^*\,a\,(a \cup b)^*$
(iii) $(a^*b^*)^*\,ab^*$
(iv) All of the above describe the language of the automaton.

Exercise 5.E
Give a deterministic finite automaton M_5 such that

$$\mathcal{L}(M_5) = \mathcal{L}(ab^*a)$$

Write the automaton by giving the tuple $M_5 = \langle Q, \Sigma, q_0, F, \delta \rangle$.

It is general knowledge that any deterministic finite automaton can be translated into a right linear grammar, as we have seen and done for M_2.

Theorem 5.6
For every deterministic finite automaton M, a right linear grammar G can be constructed such that $\mathcal{L}(G) = \mathcal{L}(M)$.
(The language that G generates is the same as the language that M accepts.) A direct result is that the language $\mathcal{L}(M)$ of a deterministic finite automaton M is always regular.

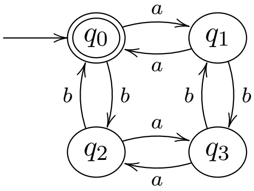
Exercise 5.F
Construct a right linear grammar for the deterministic finite automaton M_1, similarly as one was constructed for M_2 above. After that, optimize the grammar by removing useless symbols as explained in Section 4.5.

Theorem 5.6 can be very useful as sometimes it is easier to create a deterministic finite automaton for a language and then derive the corresponding right linear context-free grammar, then derive a context-free grammar directly.

Example 5.7
Let us consider the language

$$L_{17} := \{w \in \{a,b\}^* \mid w \text{ contains an even number of } a\text{'s and an even number of } b\text{'s}\}$$

Creating a context-free grammar for L_{17} can be considered somewhat difficult, but creating an automaton that accepts the language is easy:



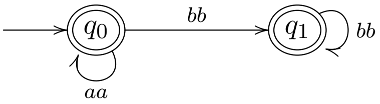
Here, q_0 represents the state that the number of a's and b's are both even, q_1 the state that the number of a's is odd and the number of b's is even, q_2 the state that the number of a's is even and the number of b's is odd, and q_3 the state that the number of a's and b's are both odd. Now this automaton can easily be transformed into a right linear context-free grammar, which is of course by definition a context-free grammar.

$$\begin{array}{rcl} S & \rightarrow & aA \mid bB \mid \lambda \\ A & \rightarrow & aS \mid bC \\ B & \rightarrow & aC \mid bS \\ C & \rightarrow & aB \mid bA \end{array}$$

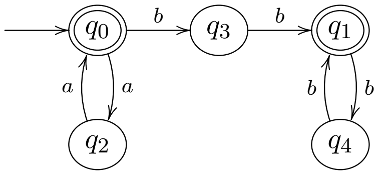
We can also translate the other way around: constructing a finite automaton for a given right linear grammar. Take a look at the right linear grammar G_{11}:

$$\begin{array}{rcl} S & \rightarrow & aaS \mid bbB \mid \lambda \\ B & \rightarrow & bbB \mid \lambda \end{array}$$

First, we introduce a state for every nonterminal, and make transitions labeled with letter sequences instead of just letters. Each nonterminal that leads to λ becomes a final state, and S becomes the initial state.

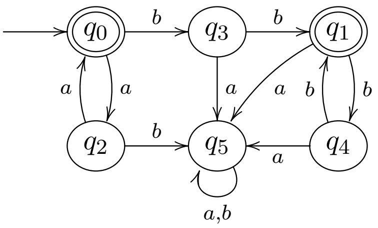


Then, we expand the letter sequences into single letter transitions by adding intermediate states, and get:



And now we are almost done. The remaining problems lies in the fact that, in a full automaton, *every state and letter* must have an outgoing transition leading to a state, which is not the case as of yet, as q_1, q_3, and q_4 do not have an a-transition, and q_2 does not have a b-transition. To solve this, we add a so-called "sink" that catches all additional useless transitions, and that any computation cannot escape from into a final state any more. This gives us our final automaton, where q_5 is the newly added sink:

M_6 :

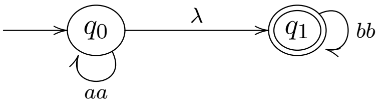


Remark 5.8
Here, we conveniently drew a single arrow, from q_5 to itself, with two labels, that actually stands for two arrows with the respective labels, because else the drawing would become a bit of a mess. Instead of adding all these arrows, we could have agreed upon omitting them, as well as the "sink." We don't do so, however, because a word such as *bba* should clearly be *not* accepted by M_6, which is definitely the case in the final automaton, but without the added arrows and sink as in the earlier version, it would halt in the final state q_1 (with remaining input), which is a bit unclear ...

This procedure of constructing automata for right linear grammars works well in general, but not in the case of production rules of the shape $S \to B$, where the right hand side doesn't contain any letters in front of the nonterminal. Then, the word in front of the nonterminal is λ, as in the case of grammar G_{12}:

$$\begin{array}{rcl} S & \to & aaS \mid B \\ B & \to & bbB \mid \lambda \end{array}$$

Performing the first step of constructing a corresponding automaton, we get:



And we end up with a transition labeled λ, the empty word, which evidently leads to a problem in the next step. The general solution is to first create an *equivalent* right linear grammar without any productions of this shape. This is always possible, but we won't show how to do this in general. For the grammar G_{12}, this initial step would lead to G_{11} which we have seen before, so that the automaton M_6 indeed accepts the language generated by G_{12}. (And G_{11} and G_{12} generate the same language: check this yourself!)

The procedure of constructing automata for right linear grammars can also fail for another reason. Take for instance the grammar G_{13} defined by

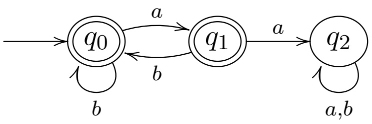
$$\begin{array}{rcl} S & \to & aS \mid aaA \mid B \\ A & \to & aA \mid B \mid \lambda \\ B & \to & bB \mid \lambda \end{array}$$

If we apply the standard algorithm and fix the aa-transition by adding an intermediate state, we would get an initial state q_0 that has two outgoing a-transitions, which is not allowed in a deterministic automaton. Again, one way to solve this problem is by trying to find an equivalent grammar that doesn't lead to multiple a-transitions from a single state. Another way to solve it would be to transform the given grammar G_{13} into a *non-deterministic automaton*, which will be defined in Section 5.3 and then apply the algorithm from Section 5.4.1 to transform it into a deterministic automaton.

However, although it may not always be easy, it is always possible to find a deterministic finite automaton that matches the right linear context-free grammar:

Theorem 5.9
For every right linear grammar G, a finite automaton M can be constructed such that $\mathcal{L}(M) = \mathcal{L}(G)$.

Example 5.10
A direct consequence of Theorem 5.9 is that we can extend the four ways of describing the language L_{14} of all words over $\{a, b\}^*$ that don't contain aa that we have seen in Example 4.38 with a fifth way. Besides describing L_{14} with natural language, with set notation, with a regular expression and with a context-free right linear grammar, we can also describe this language with a deterministic finite automaton:

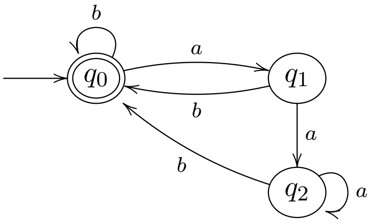


Exercise 5.G
Construct a deterministic finite automaton that recognizes the language of the following grammar G_{14}:

$$\begin{array}{lcl} S & \rightarrow & abS \mid aA \mid bB \\ A & \rightarrow & aA \mid \lambda \\ B & \rightarrow & bB \mid \lambda \end{array}$$

Exercise 5.H
Consider the deterministic finite automaton M_7:

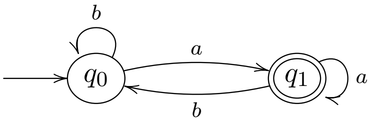
M_7 :



Construct a right linear grammar that generates $\mathcal{L}(M_7)$.

Exercise 5.I
Consider the deterministic finite automaton M_8:

M_8 :



(i) Construct a right linear grammar that generates $\mathcal{L}(M_8)$.
(ii) Provide a description of $\mathcal{L}(M_8)$. Try to make it as simple as possible.
(iii) If all states are made into final states, which language does M_8 recognize?
(iv) If we swap the final states with non-final states (every final state becomes a non-final state and the other way around), which language would M_8 recognize?

Exercise 5.J
Define $L_{18} := \{(ab)^k\,(aba)^l \mid k, l \geq 0\}$ over the alphabet $\Sigma = \{a, b\}$. Construct a deterministic finite automaton that recognizes this language.

Exercise 5.K
Define $L_{19} := \{(ab)^k\,x\,(ab)^l \mid x \in \{a, b\}, k, l \geq 0\}$ over the alphabet $\Sigma = \{a, b\}$. Construct a deterministic finite automaton that recognizes this language.

Exercise 5.L
In this exercise we consider the class of deterministic finite automata ${M_9}^i$ over alphabet $\Sigma = \{a, b\}$, where i is a natural number representing an index, such that

- ${M_9}^i$ has at most two states,
- $\lambda \notin \mathcal{L}\left({M_9}^i\right)$, and
- $\mathcal{L}\left({M_9}^i\right) \neq \varnothing$

(i) How many different deterministic finite automata exist with these properties?
(ii) Draw all these automata.
(iii) For each of these automata ${M_9}^i$, provide a regular expression that generates the same language $\mathcal{L}\left({M_9}^i\right)$.
(iv) How many different languages do these automata generate?

5.3 Non-deterministic automata

In the definition of an automaton we had a special restriction:

> For each state q and each symbol a from the alphabet, there is *exactly one arrow* from q with label a.

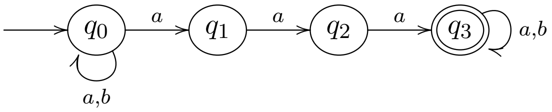
We can weaken this restriction in the following way:

- From each state q there are *finitely many* arrows with label a for each $a \in \Sigma$ (i.e., 0, 1 or more).
- From each state q there are *finitely many* arrows with label λ (i.e., 0, 1 or more).

We call an automaton in which we allow this a *non-deterministic finite automaton* or *NFA*.

Example 5.11
Consider the following automaton

M_{10} :



(i) If we take the word *baaa* as input, there are four possible computations depending on which arrow we choose for the a's:

$$\begin{array}{l} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0 \\ q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_1 \\ q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_2 \\ q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_3 \end{array}$$

So we can end in q_0, q_1, q_2 or in q_3. The last is a final state, the others are not. This phenomenon we call *non-determinism*: the automaton executes *non-deterministically* (in a way that can not be determined beforehand) one of the possible computations.

(ii) If we take the word *bbb* as input, only one computation is possible, which ends in q_0.

$$q_0 \xrightarrow{b} q_0 \xrightarrow{b} q_0 \xrightarrow{b} q_0$$

(iii) With input *baa* three computations are possible, that all end in a non-final state.

$$\begin{array}{l} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0 \\ q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_1 \\ q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_2 \end{array}$$

(iv) With input *aba* three computations are possible.

$$\begin{array}{l} q_0 \overset{a}{\to} q_0 \overset{b}{\to} q_0 \overset{a}{\to} q_0 \\ q_0 \overset{a}{\to} q_0 \overset{b}{\to} q_0 \overset{a}{\to} q_1 \\ q_0 \overset{a}{\to} q_1 \end{array}$$

The first one ends in q_0, the second one ends in q_1 and the third one 'ends' in q_1 *without having processed the full word yet* (only the letter a has been read). This situation we call *deadlock*: the computation cannot continue, despite the fact that not all input has been read.

When does a non-deterministic finite automaton *accept* a word w?

Definition 5.12
The non-deterministic automaton M *accepts* the word w when with input w there *exists a computation* that ends in a final state with *the whole word having been read.*

A computation that "consumes" the whole input and ends in a final state, we also call a *successful* or *accepting* computation. So: if a computation stops in a *deadlock* it is not a successful computation.

Let us also make precise what is a non-deterministic finite automaton, and what is the language that such an automaton accepts.

Definition 5.13
A *non-deterministic finite automaton* consists of the following five components

1. A finite set Σ, the input alphabet, or set of *atomic actions*,
2. A finite set Q of *states*,
3. A distinguished state $q_0 \in Q$ called the *initial state*,
4. A set of distinguished states $F \subseteq Q$, the *final states*,
5. A *transition function* δ, that for each state q and $d \in \Sigma \cup \{\lambda\}$ gives a *set of states* $\delta(q, d)$. (If $q' \in \delta(q, d)$, then there is an arrow $q \overset{d}{\to} q'$, so these are the labeled arrows in the diagram of the automaton).

Non-deterministic finite automata are often written as a quintuple $M := \langle \Sigma, Q, q_0, F, \delta \rangle$. The *language of* M, $\mathcal{L}(M)$, is defined as follows:

$$\mathcal{L}(M) := \{w \in \Sigma^* \mid w \text{ is accepted by } M\}.$$

So: $w \in \mathcal{L}(M)$ if and only if there is a computation of the automaton M with input w that stops in a final state after having read all symbols in w.

Remark 5.14
If we look at the mathematical definitions of a deterministic finite automaton, we see that the type of the transition function δ is different from the type of the transition function δ' of a non-deterministic finite automaton. So technically, a deterministic automaton does not comply with the mathematical definition of a non-deterministic automaton. However, if one draws a deterministic finite automaton, it really looks like a simple form of a non-deterministic finite automaton that doesn't have λ-transitions and that has multiple outgoing arrows for the symbols in the alphabet. Fortunately, we can mathematically *embed* the deterministic automata easily into the set of non-deterministic automata, by adapting the transition function

in a trivial way. For a deterministic automaton the type of the transition function is $\delta : Q \times \Sigma \to Q$. For a non-deterministic automaton the type of the transition function is $\delta' : Q \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q)$[1] So given our deterministic transition function δ we can trivially define a non-deterministic transition function δ' of the proper type without changing the accepted language by defining:

$$\begin{aligned} \delta'(q,s) &= \{\delta(q,s)\} \\ \delta'(q,\lambda) &= \varnothing \end{aligned}$$

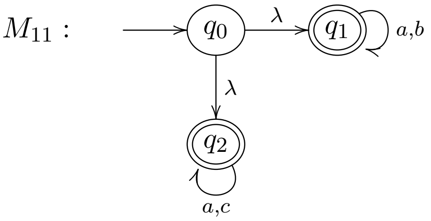
So although, technically, a deterministic finite automaton is not a non-deterministic automaton, by this construction we can justify that we *say* that in practice it is one!

Exercise 5.M

(i) Investigate in the non-deterministic finite automaton M_{10} that we defined before which computations exist with input *abaaa*, *ababa*, *ab* and *baaab*.
(ii) Which of these words are accepted?
(iii) Describe the language that M_{10} accepts.
(iv) Adapt M_{10} in such a way that it accepts $\{w \mid w \text{ ends with } aaa\}$.

Exercise 5.N

Consider the non-deterministic finite automaton M_{11}.



(i) Which computations are possible with input *aba*, *cac*, *abc* and λ?
(ii) Which of these words are accepted?
(iii) Describe the language that M_{11} accepts.

Exercise 5.O

(i) Construct a non-deterministic automaton with at most five states that accepts the language L_{18} from Exercise 5.J.
(ii) Construct a non-deterministic automaton with at most four states that accepts the language L_{19} from Exercise 5.K.

Can we do more using non-deterministic automata than with deterministic automata? Yes, we can model non-deterministic computations. But can we also accept languages that we could not accept before? In other words:

> Does there exist a language L for which we *do* have a non-deterministic automaton M that accepts L ($L = \mathcal{L}(M)$), but for which there is *no* deterministic automaton M' that accepts L ($L = \mathcal{L}(M')$)?

The answer is no:

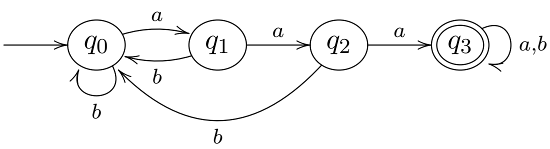
Theorem 5.15

For each non-deterministic automaton M we can make a deterministic automaton M' such that $\mathcal{L}(M) = \mathcal{L}(M')$.

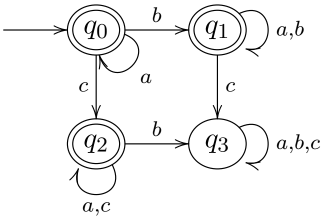
[1] The definition of $\mathcal{P}(Q)$, the power set of Q is given in Section 5.4.1.

The construction is not terribly difficult, but we will not give it here.[2] We illustrate it using two examples and then we immediately see why non-deterministic automata are sometimes useful (because they are much smaller). Consider therefore the two deterministic finite automata M_{12} and M_{13} that correspond to the non-deterministic M_{10} and M_{11} respectively. Check for yourself that these automata indeed accept identical languages.

M_{12} :



M_{13} :



Exercise 5.P

(i) Construct a non-deterministic finite automaton for the language

$$L = \{w \in \{a, b, c\}^* \mid w \text{ ends with } aab \text{ or } w \text{ ends with } ccb\}$$

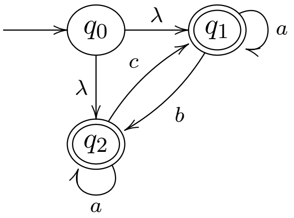
(ii) Construct a non-deterministic finite automaton for the language

$$L' = \{w \in \{a, b, c\}^* \mid w = vu \text{ and } v \text{ contains } aa \text{ and } u \text{ contains } bb\}$$

Exercise 5.Q

(i) Describe the language that automaton M_{14} accepts.

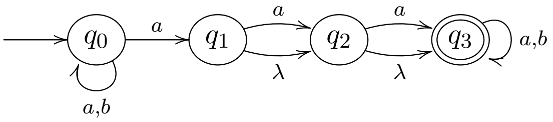
M_{14} :



(ii) Construct a deterministic finite automaton that accepts the language of M_{14}.

(iii) Describe the language that M_{15} accepts.

M_{15} :



(iv) Construct a deterministic finite automaton that accepts the language of M_{15}.

Exercise 5.R

(i) Suppose M_1 is a finite automaton that accepts L_1 and that M_2 is a finite automaton that accepts L_2. Construct a non-deterministic finite automaton that accepts $L_1 \cup L_2$. [*Hint:* Look at example M_{11} above.]

(ii) Prove that the class of regular languages is closed under $\cup$, i.e., if L_1 and L_2 are regular, then $L_1 \cup L_2$ is also regular.

[2] We'll present it later on in Section 5.4.1.

(iii) Suppose M_1 is a finite automaton that accepts L_1 and M_2 is a finite automaton that accepts L_2. Define a non-deterministic finite automaton that accepts L_1L_2. Remember that L_1L_2 is the language that consists of first a word from L_1 and then a word from L_2, so $L_1L_2 = \{vw \mid v \in L_1, w \in L_2\}$.

(iv) Prove that the class of regular languages is closed under *concatenation*, i.e., if L_1 and L_2 are regular, then L_1L_2 is also regular.

Remark 5.16
In the last two chapters we have introduced five formalisms:

1. regular expressions
2. context-free grammars
3. right linear context-free grammars
4. deterministic finite automata
5. non-deterministic finite automata

If we look at the languages that can be described or generated by these formalisms, we see that four of these five formalisms actually describe the same set of languages, namely the *regular languages*. Only the class of *context-free languages* is really larger. See Figure 5.1 for the relations. Each arrow implies 'inclusion'. So the set of languages in the box at the start of an arrow is included in the set of languages in the box at the end of the arrow by means of the given label.

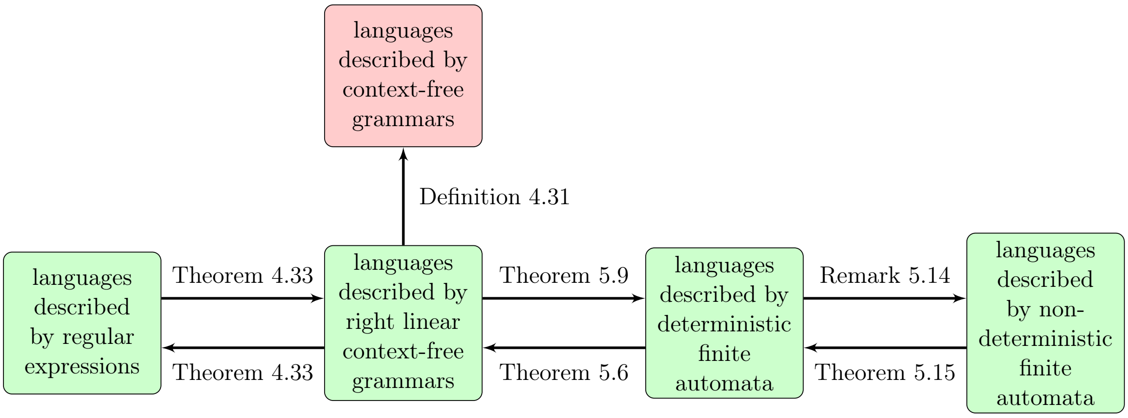

languages described by context-free grammars
Definition 4.31
languages described by regular expressions
Theorem 4.33
Theorem 4.33
languages described by right linear context-free grammars
Theorem 5.9
Theorem 5.6
languages described by deterministic finite automata
Remark 5.14
Theorem 5.15
languages described by non-deterministic finite automata

Figure 5.1: Five formalisms

Note that the set of languages described by context-free grammars is strictly larger than the set of languages described by right linear context-free grammars. As an example we can take language L_2 that we defined in Chapter 4:

$$L_2 = \{a^n b^n \mid n \in \mathbb{N}\}$$

This is the default example of a non-regular language, so there doesn't exists a right linear context-free grammar that describes it, but there does exist a quite trivial general context-free grammar that describes it:

$$S \to aSb \mid \lambda$$

Note that it is not right linear as the S in $S \to aSb$ is not at the far right. The proof[3] that L_2 is not regular goes beyond the scope of this course. It uses the *pumping lemma for regular languages*.

[3] See https://en.wikipedia.org/wiki/Pumping_lemma_for_regular_languages.

Remark 5.17
Besides the five (or two!) classes of languages in Figure 5.1, there exist more classes. One of them is the class of languages that cannot be described by a context-free grammar. The classical example for such a language is pretty similar to L_2:

$$L_{20} := \{a^n b^n c^n \mid n \in \mathbb{N}\}$$

The proof[4] that L_{20} is not context-free uses the *pumping lemma for context-free languages.*

5.4 Two algorithms that you don't have to know by heart for the exam

In this section we describe two algorithms that are not really part of the course objectives, but that are usually discussed in one of the lectures anyway. For reasons of completeness we added these algorithms in this section. You don't have to know the details of how these algorithms work, but you should know that they exist and what they do.

5.4.1 Converting an NFA to a DFA

Directly after Theorem 5.15 we stated that the conversion from an NFA to a DFA is not 'terribly difficult'. Here we will show you how it works by means of an example.

Note that this algorithm is usually called the *powerset construction.* So let us first introduce the concept of a powerset. In short: the powerset of a given set A, denoted as $\mathcal{P}(A)$, is a set containing all subsets of set A.

Example 5.18
Let A be the set $\{\text{Freek}, \text{Engelbert}\}$. Then A has four subsets:

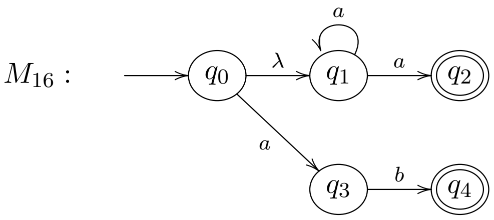
- none of the two elements are included in the subset,
- only Freek is included in the subset,
- only Engelbert is included in the subset, or
- both elements Freek and Engelbert are included in the subset.

Hence we get

$$\mathcal{P}(A) = \{ \quad \varnothing, \quad \{\text{Freek}\}, \quad \{\text{Engelbert}\}, \quad \{\text{Freek}, \text{Engelbert}\} \quad \}$$

Remark 5.19
For all sets A we have that $\varnothing \in \mathcal{P}(A)$ and $A \in \mathcal{P}(A)$. Note that we wrote '$\in$' and not '$\subseteq$'!

Example 5.20
Let us consider the following NFA.



[4] See https://en.wikipedia.org/wiki/Pumping_lemma_for_context-free_languages.

We can describe this automaton by this quintuple:

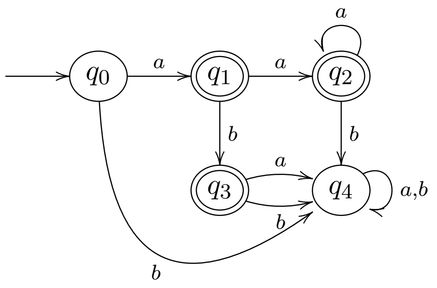
$$\langle \Sigma, Q, q_0, F, \delta \rangle$$

where $\Sigma = \{a, b\}$, $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_2, q_4\}$, and the transition function δ of type $Q \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q)$ is defined by:

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
\delta(q_0, a) & = & \{q_3\} & \delta(q_0, b) & = & \varnothing & \delta(q_0, \lambda) & = & \{q_1\} \\
\delta(q_1, a) & = & \{q_1, q_2\} & \delta(q_1, b) & = & \varnothing & \delta(q_1, \lambda) & = & \varnothing \\
\delta(q_2, a) & = & \varnothing & \delta(q_2, b) & = & \varnothing & \delta(q_2, \lambda) & = & \varnothing \\
\delta(q_3, a) & = & \varnothing & \delta(q_3, b) & = & \{q_4\} & \delta(q_3, \lambda) & = & \varnothing \\
\delta(q_4, a) & = & \varnothing & \delta(q_4, b) & = & \varnothing & \delta(q_4, \lambda) & = & \varnothing
\end{array}$$

Clearly, this automaton accepts the language $\{ab, a, aa, aaa, aaaa, aaaaa, \ldots\}$. Creating a DFA from scratch for this language requires some thinking, but it is not that difficult:

${M_{16}}'$:



However, for more difficult languages like the one in Exercise 5.J it can be convenient to use an algorithm that automatically derives the corresponding DFA given an NFA.

As said before, the construction is called the powerset construction. This is because the states in the final DFA are in fact subsets of the set of all states in the original NFA. Given the fact that M_{16} has five states $\{q_0, q_1, q_2, q_3, q_4\}$, we know that the constructed corresponding DFA has at most $2^5 = 32$ states, each labeled with a subset:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{P}(\{q_0, q_1, q_2, q_3, q_4\})} \\
= & \{ \quad \varnothing, \\
& \{q_0\}, \{q_1\}, \{q_2\}, \{q_3\}, \{q_4\}, \\
& \{q_0, q_1\}, \{q_0, q_2\}, \{q_0, q_3\}, \{q_0, q_4\}, \\
& \{q_1, q_2\}, \{q_1, q_3\}, \{q_1, q_4\}, \\
& \ldots \\
& \{q_0, q_1, q_2, q_3\}, \{q_0, q_1, q_2, q_4\}, \{q_0, q_1, q_3, q_4\}, \{q_0, q_2, q_3, q_4\}, \{q_1, q_2, q_3, q_4\}, \\
& \{q_0, q_1, q_2, q_3, q_4\} \quad \}
\end{array}$$

Fortunately, we don't need all these states. The algorithm only creates the states that are really needed.

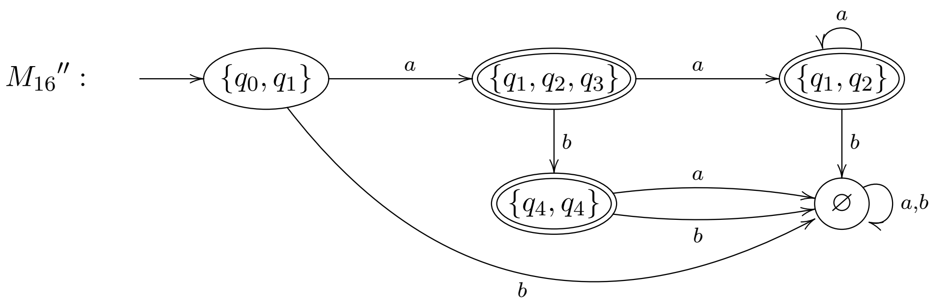
The main idea is that the states in our derived DFA are subsets of the original set of states, in such a way that they are *closed under λ-transitions*. This last thing means that if there is a state q in the subset identifying a state, and in the original NFA there is a λ-transition to a state q', then also q' must be in this same subset identifying the state.

This is the algorithm:

- Determine the initial state of our new DFA. It is the subset containing q_0 and all other states that you can reach from q_0 by using λ-transitions in M_{16}. So in this case we get as initial state the state labeled $\{q_0, q_1\}$.

- Next, for each symbol of the alphabet Σ, we collect all the states we can get to from each of the states in the subset.
- So from the initial state $\{q_0, q_1\}$ reading an a we get to the state $\{q_1, q_2, q_3\}$, because the transition function δ gives $\delta(q_0, a) = \{q_3\}$ and $\delta(q_1, a) = \{q_1, q_2\}$.
- And from the initial state $\{q_0, q_1\}$ reading a b we get to the state $\varnothing$, since the δ indicates that there are no b-transitions from q_0 and q_1.
- We repeat this for all new states. From $\{q_1, q_2, q_3\}$ reading an a we get to $\{q_1, q_2\}$.
- And from $\{q_1, q_2, q_3\}$ reading a b we get to $\{q_4\}$.
- From $\varnothing$ we cannot get anywhere reading an a or a b, so this state is a natural sink!
- From $\{q_1, q_2\}$ reading an a we get to $\{q_1, q_2\}$.
- From $\{q_1, q_2\}$ reading a b we get to $\varnothing$.
- From $\{q_4\}$ reading an a we get to $\varnothing$.
- From $\{q_4\}$ reading a b we get to $\varnothing$.
- Since we didn't create any new states anymore, we have all our states.
- Now all that is left to do is to determine the final states. Final states are the states identified by subsets that contain at least one of the original final states of our NFA M_{16}.
- Hence the states labeled $\{q_1, q_2, q_3\}$ (because of q_2), $\{q_1, q_2\}$ (because of q_2), and $\{q_4\}$ (because of q_4) must be final states in our new DFA.

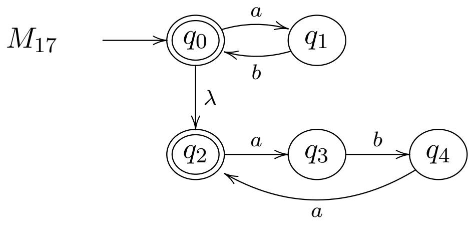
Now that we have all information for our DFA we can draw it:



Apart from the names of the states, we see that we get exactly the automaton M_{16}'.

Now let us apply this algorithm on a more difficult automaton:

Example 5.21

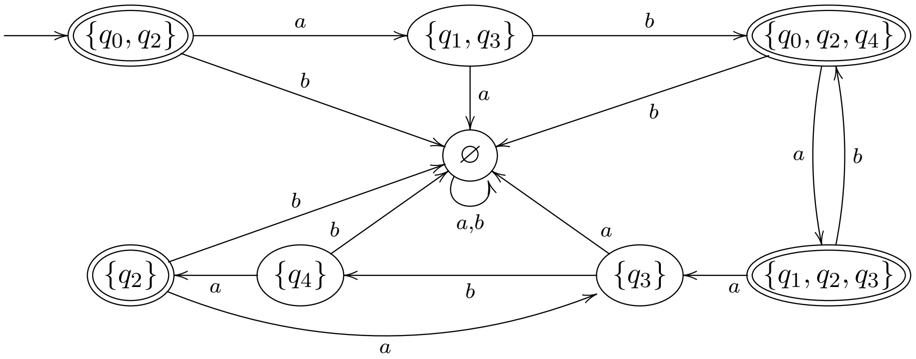


Applying the algorithm gives:

- The initial state will be $\{q_0, q_2\}$.

- From $\{q_0, q_2\}$ reading an a we go to $\{q_1, q_3\}$.
- From $\{q_0, q_2\}$ reading a b we go to $\varnothing$.
- From $\{q_1, q_3\}$ reading an a we go to $\varnothing$.
- From $\{q_1, q_3\}$ reading a b we go to $\{q_0, q_2, q_4\}$.
- From $\{q_0, q_2, q_4\}$ reading an a we go to $\{q_1, q_2, q_3\}$.
- From $\{q_0, q_2, q_4\}$ reading a b we go to $\varnothing$.
- From $\{q_1, q_2, q_3\}$ reading an a we go to $\{q_3\}$.
- From $\{q_1, q_2, q_3\}$ reading a b we go to $\{q_0, q_2, q_4\}$.
- From $\{q_3\}$ reading an a we go to $\varnothing$.
- From $\{q_3\}$ reading a b we go to $\{q_4\}$.
- From $\{q_4\}$ reading an a we go to $\{q_2\}$.
- From $\{q_4\}$ reading a b we go to $\varnothing$.
- From $\{q_2\}$ reading an a we go to $\{q_3\}$.
- From $\{q_2\}$ reading a b we go to $\varnothing$.
- From $\varnothing$ reading an a we go to $\varnothing$.
- From $\varnothing$ reading a b we go to $\varnothing$.
- The final states are: $\{q_0, q_2\}$, $\{q_0, q_2, q_4\}$, $\{q_1, q_2, q_3\}$, and $\{q_2\}$.

This gives the DFA



Now go and compare this automaton with your solution to Exercise 5.J!

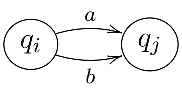
5.4.2 Converting an NFA to a regular expression

In Theorem 5.6 we have seen that for every DFA M a right linear grammar G can be constructed such that $\mathcal{L}(G) = \mathcal{L}(M)$. And in particular it followed that the language $\mathcal{L}(M)$ is always regular. Hence there must exist a regular expression r such that $\mathcal{L}(r) = \mathcal{L}(M)$.

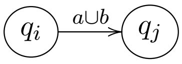
Here we will present an algorithm that actually derives this expression.

Although Theorem 5.6 is only about deterministic finite automata, it also holds for non-deterministic automata, since we have just seen how we can convert an NFA into a DFA accepting the same language. Therefore, our algorithm actually computes a regular expression given an NFA. Since any DFA is also an NFA, this is no restriction whatsoever.

The main idea in this algorithm is that we no longer use symbols from the alphabet as labels in our diagrams (we call them ‘diagrams’ and not ‘automata’, because they are neither NFAs or DFAs), but regular expressions. And there will just be one arrow in the same direction between two states. So if our original automaton contains

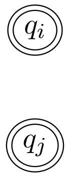


that will become

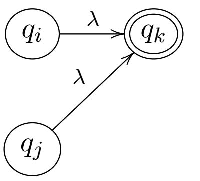


in our diagram.

In addition, our diagrams will only have exactly one final state. Note that this can easily be achieved by adding λ-transitions from all existing final states in the automaton (which then no long will be final) to a new single final state. So something like

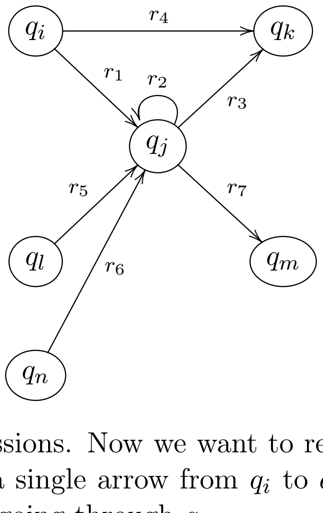


becomes



And once we have a diagram with one initial state and one final state, the reduction process starts. The idea is that in every step we remove a single state which is not the initial state and not the final state, by combining regular expressions, until we only have the initial state and the final state left, which can actually be the same. So in the end we have at most two states.

Assume we have the following situation in our diagram:



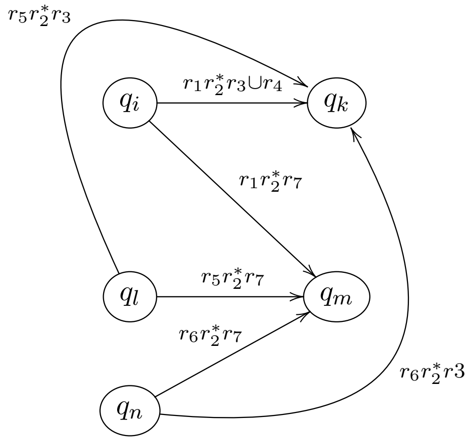
where all the r_i are regular expressions. Now we want to remove the state q_j. So we want to end up with a diagram that has a single arrow from q_i to q_k, representing all possibilities to get from q_i to q_k with or without going through q_j.

Likewise we want to get single arrows from q_i to q_m, q_l to q_k, q_l to q_m, q_n to q_k and q_n to q_m. Basically, for every combination of states that are connected via q_j we have to have an arrow in the next diagram.

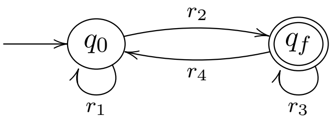
Note that we can get from q_i to q_k by expression r_4 directly, so this must be part of the new label for the arrow from q_i to q_k. But we can also go via q_j, which implies we first have regular expression r_1, then zero or many times r_2 and then once r_3. So the label for the arrow from q_i to q_k should be

$$r_1 r_2^* r_3 \cup r_4$$

Doing the same thing for the other states involved, our diagram can be reduced to



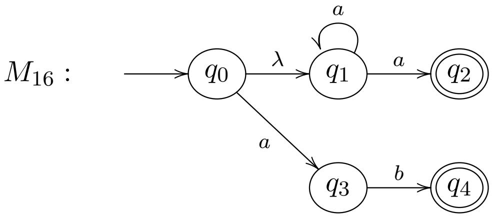
If we continue removing states until only the initial and final state are left, we typically end up with a diagram that looks like this:



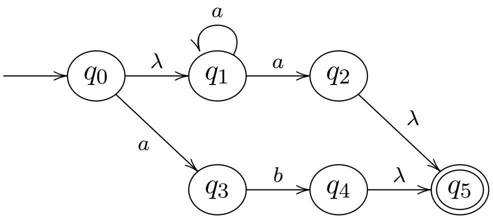
The corresponding regular expression can now be derived easily from the diagram:

$$r_1^* r_2 (r_3 \cup r_4 r_1^* r_2)^*$$

Example 5.22
Recall our automaton

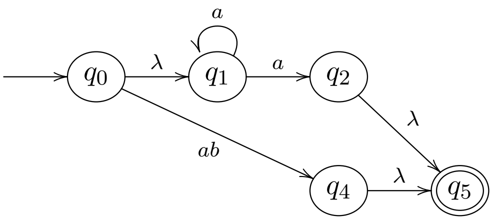


First we have to fix the amount of final states. This gives us:

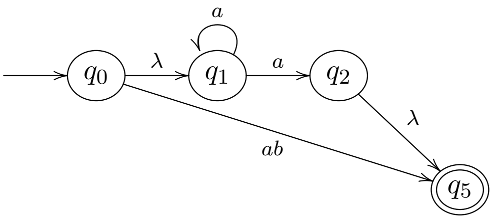


Now we have to replace our symbols by the corresponding regular expressions. Which gives us exactly the same diagram! (So we won't draw it again.)

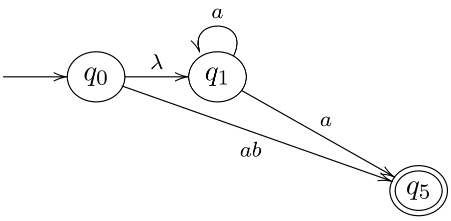
Removing q_3 gives the following diagram:



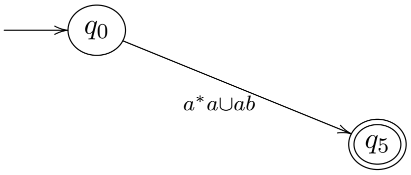
And removing q_4 gives:



Removing q_2 gives:



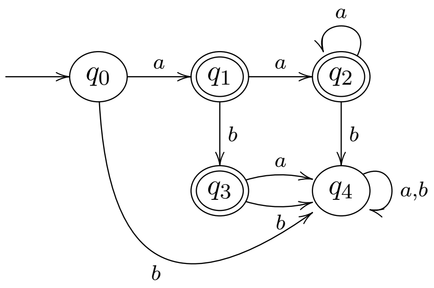
And finally removing q_1 gives:



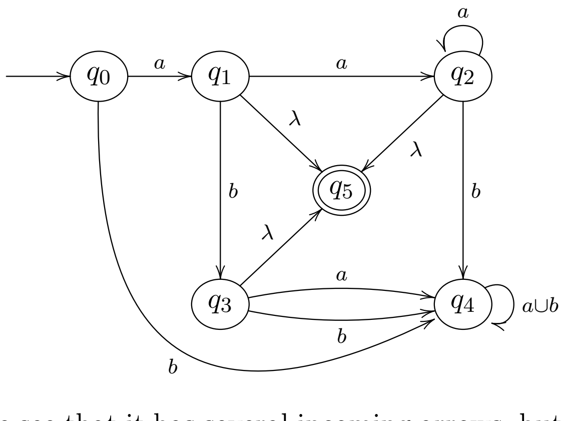
So $\mathcal{L}(M_{16}) = \mathcal{L}(a^*a \cup ab)$.

Example 5.23
Now let us see what would have happened if we started with our DFA for the same language:

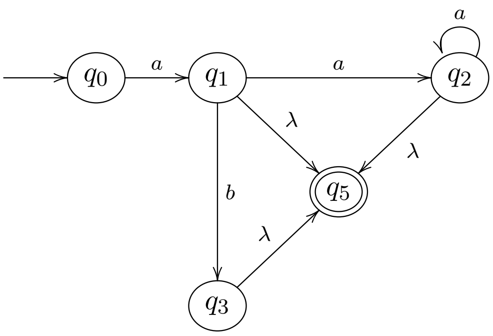
$M_{16}{}'$:



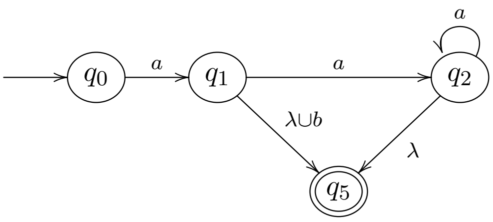
Let us first make sure that there is only one final state and that we have regular expressions as labels. This gives the diagram:



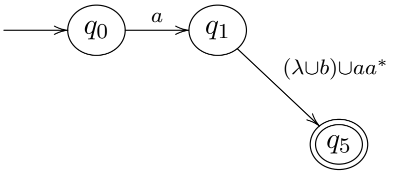
Now let us remove q_4. We see that it has several incoming arrows, but the only outgoing arrow goes back to q_4 again. So if we remove q_4, we don't have to add any new arrows! We can simply remove it, because it has no affect on accepting words. Which is of course what can be expected from a sink! So we get:



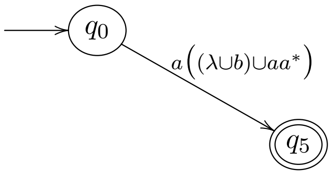
Now if we remove q_3 we get:



And if we remove q_2 from this diagram we get:



Finally removing q_1 gives:



So we get $\mathcal{L}(M_{16}{}') = \mathcal{L}\left(a\big((\lambda \cup b) \cup aa^*\big)\right) = \mathcal{L}(aa^* \cup ab)$. Note that the order of reduction is relevant for the actual expression. Can you find an order of reduction that leads to the regular expression $a \cup aaa^* \cup ab$?

5.5 Important concepts

Chapter 6

Modal logic

Although propositional logic and predicate logic are the most well-known, there are many other 'logics'. Different logics vary in their ability to represent nuances of statements. They differ in which operators are used to write down formulas, and sometimes they differ in the way interpretations are given to those operators as well.

Some examples are: Floyd-Hoare logic (in which you can describe accurately, the behavior of programs), intuitionistic logic (which expresses exactly those things that can be computed by computers), and paraconsistent logic (in which it is possible to retract the truth statements as new information becomes available).

A special class of logics are the so-called *modal logics*, which are used to describe *to which extent* we should believe certain statements to be true. Modal logics are the subject matter of this chapter.

Within the class of modal logics, there is the important sub-class of *temporal logics*. These logics are able to express *at which points in time* statements are true. At the end of this chapter, we present a temporal logic in more detail. This chapter is partially based on [2, Chapter 5], so that book is a good source for further reading about modal logic. In addition, we recommend the book [4].

6.1 Necessity and possibility

Traditionally, modal logics express the notion of *necessity*. Take the following sentence:

It is raining.

This sentence could be true or not true, but it is not *necessarily* true. It does not always rain, and even when it does, the reason is not simply that it *must* be raining: if the weather was different it may not have rained. The following sentence however, is necessarily true:

If it rains, water falls from the sky.

The necessity of the truth of this sentence follows from the meaning of the word 'raining', which is exactly, that water falls from the sky.

This difference of necessity does not find expression in propositional logic. If we use the following dictionary:

R	it is raining
W	water is falling from the sky

then the formalizations of the sentences are:

$$R$$

and

$$R \to W$$

in which the necessity is not clearly expressed. To this purpose, modal logic introduces a new operator $\Box$, that should be read as meaning 'necessarily.' The sentence:

It is necessarily true that raining implies that water falls from the sky.

is then written formally as:

$$\Box(R \to W)$$

You might think the reverse is true as well, that water falling from the sky implies that it is raining, but this is not true. For instance, turning on a garden hose, or standing under a waterfall, are situations in which water does fall from the sky, without it raining. Therefore, we have:

It is not necessarily true that water falling from the sky implies that it is raining.

which, written as a formula in modal logic, is:

$$\neg\Box(W \to R)$$

Of course it is still possible for

$$W \to R$$

to hold in certain situations. For example, if it is raining, then the right-hand side of the implication is true, and hence (as we can see in the truth table for implication) the whole formula is true.

Besides necessity, modal logic also has a notation for *possibility*. This is written as: $\Diamond$. The true sentence:

It is possible that it is raining.

is then expressed in modal logic by the formula:

$$\Diamond R$$

If you think about it a bit, you will find that the above statement means the same as:

It it not necessary for it not to be raining.

Which is in turn expressed as:

$$\neg\Box\neg R$$

Remark 6.1
This phenomenon is called *conjugation* and it appears in many variations. In modal logic we have $\Diamond f \equiv \neg\Box\neg f$, $\neg\Diamond f \equiv \Box\neg f$, $\Box f \equiv \neg\Diamond\neg f$, and $\neg\Box f \equiv \Diamond\neg f$. And the De Morgan laws, both in propositional logic and in predicate logic, are also examples of conjugation.

Exercise 6.A
So we see that the formula $\Diamond R$ means the same as $\neg\Box\neg R$. Try to find a formula without the symbol '$\Box$' that means the same as $\Box R$. Then, translate both $\Box R$ and the formula you found to ordinary English.

Definition 6.2
Any statement U will either be:

- necessary
- impossible
- contingent

A statement is said to be *contingent* when it is not necessarily true, nor impossibly true. That is, its truth is somewhere between necessarily true and necessarily false.

Exercise 6.B
In modal logic the statement 'U is necessarily true' is symbolically represented as $\Box U$, and 'U is impossible' as $\Box\neg U$, or alternatively, $\neg\Diamond U$. Give two different formulas in modal logic which express the statement that 'U is contingent.'

Exercise 6.C
Use the following dictionary

M	I have money
B	I am buying something

to translate these English sentences to formulas of modal logic:
(i) *It is possible for me to buy something without having money.*
(ii) *It is necessarily true, if I am buying something, for me to have money.*
(iii) *It is possible that if I buy something I don't have any money.*
Which of *these* sentences seem true? Explain why.

6.2 Syntax

We will now give a context-free grammar of the formulas of modal logic. (Modal predicate logic also exists, but we will not treat it here.) This grammar is exactly the same as the one we have given for propositional logic before (see Remark 1.5), except that the two modal operators are added:

$$S \ \rightarrow\ a \mid \neg S \mid (S \wedge S) \mid (S \vee S) \mid (S \rightarrow S) \mid (S \leftrightarrow S) \mid \Box S \mid \Diamond S$$

Convention 6.3
Just as with propositional logic, we allow ourselves to leave out unnecessary parentheses. In doing this, we stick to the convention that the modal operators have the same operator precedence as negation '¬', which means that they bind stronger than the binary operators.

This means that the formula

$$\Diamond a \wedge b \rightarrow \Diamond(\Box a \vee \neg c)$$

should be read as

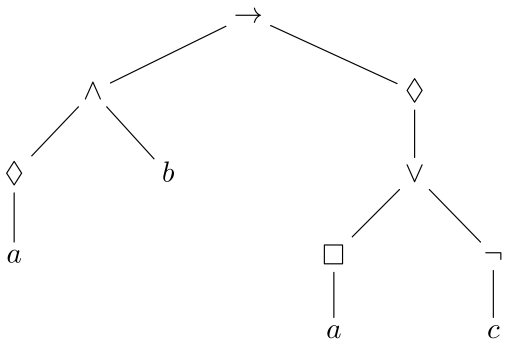
$$(((\Diamond a) \wedge b) \rightarrow (\Diamond((\Box a) \vee (\neg c))))$$

Because the grammar does not add parentheses to the unary operators, the following formula is the 'official' form of the ones above:

$$((\Diamond a \wedge b) \rightarrow \Diamond(\Box a \vee \neg c))$$

The first two should be seen as different (and possibly clearer) ways to represent this 17-symbol-long formula.

Also for modal logic we can express the structure of the formula in a parse tree:



As usual, in the tree the atomic propositions are the leaves and the logical operators the nodes.

Exercise 6.D
For each of the formulas below, give the 'official' form according to the grammar of modal logic, and also draw a tree according to its structure.

(i) $\Box\,(\Box a)$
(ii) $\Box a \to a$
(iii) $(\Box a) \to a$
(iv) $\Box\,(a \to a)$
(v) $\neg a \to \neg\Box a$
(vi) $\Diamond a \to \Box\Diamond a$
(vii) $\Box a \to \Diamond a$
(viii) $\Diamond a \wedge b \to \Diamond\Box a \vee \neg c$

6.3 Modalities

Up until now the operator $\Box$ has had the fixed meaning of ‘necessity‘ and the operator $\Diamond$ that of ‘possibility’. This is the traditional reading, but only one of a variety of possible interpretations of these symbols. Depending on what we interpret the symbols to mean, we get different modal logics:

logic	modality	f	$\Box f$	$\Diamond f$
modal logic	necessity	f is true	f is necessarily true; f is true in all possible worlds	f is possibly true; f is true in some possible worlds
epistemic logic	knowledge	f is true	I know that f	f doesn't contradict my knowledge; I don't know that f is false
doxastic logic	belief	f is true	I believe that f	f doesn't contradict my beliefs; I don't believe that f is false
temporal logic	time	f is now true	f is always true	f is sometimes true
deontic logic	obligation	I do f	f ought to be done; I must do f	f is permissible; I may do f
dynamic logic	(non-deterministic) programs	f holds	after any execution of p, f holds	after some execution of p, f holds

Note that for dynamic logic there is a p in the description. That is because it is important to know about which program one is reasoning. Therefore, often one writes $\Box_p f$ and $\Diamond_p f$.

Example 6.4
Consider the dictionary

S	it is Sunday
C	I go to church

and the sentence

On Sunday I always go to church

There are two ways to translate it into a formula:

1. $S \to \Box C$, which means: *If it is Sunday now, then I always go to church.* So in particular, also on next Wednesday.
2. $\Box(S \to C)$, which means: *It is always the case that if it is Sunday now, then I go to church.*

Which formula do you think is more appropriate?

If not specified otherwise, 'modal logic' often refers to the traditional reading of 'necessity'. At the same time, it is also used to refer to the whole class of modal logics.

Exercise 6.E
Use the following dictionary:

R	I am ready
H	I am going home

For each of the logics listed in the table above (except program-logic), give an English translation of the formula:

$$\neg R \to \neg\Diamond H$$

Each of the logics enumerated in the table, have yet again their own variations. For instance, just as there are a whole set of modal logics, so too, are there many different temporal logics. The meaning 'true at all times' can be changed to 'always true from this point on' or maybe 'always true after this point in time.' Or else, it could take into account that it is not yet clear what is going to happen. (This is called 'branching time'.)

6.4 Axioms

Depending on what the modal operators are meant to describe, you might want them to have different properties. For instance, whether the formula

$$\Box f \to f$$

should always be true. (Note: this should be read as '$(\Box f) \to f$' and not as '$\Box(f \to f)$', because the $\Box$ binds stronger than the $\to$.) In the traditional interpretation of modal logic this indeed always holds, because it translates to:

If f is necessarily true, then f holds.

And of course, necessary truth implies truth. But you wouldn't want it to hold in doxastic logic, because then it would be an encoding of the following English sentence:

If I believe f to be true, then f holds.

Although we all know that belief does not imply truth, because a belief may be mistaken. If we turn to the logic of knowledge, epistemic logic, we want it to be true again:

If I know that f holds, then f holds.

Because *knowing* something to be true implies in particular that it must indeed be true (in addition to the fact that it is known). Hence, we see that the truth of a particular property depends on the chosen interpretation of $\Box$. Do you think it holds in deontic or temporal logic?

In modal logic, there are many such principles (properties), called *axiom schemes* of modal logic. Some of the most important ones are listed below:

name	axiom scheme	property
K	$\Box(f \rightarrow g) \rightarrow (\Box f \rightarrow \Box g)$	distributive
T	$\Box f \rightarrow f$	reflexive
B	$f \rightarrow \Box\Diamond f$	symmetric
4	$\Box f \rightarrow \Box\Box f$	transitive
5	$\Diamond f \rightarrow \Box\Diamond f$	Euclidean
D	$\Box f \rightarrow \Diamond f$	serial

The first column presents the letter or number usually used to denote the axiom scheme, and the third column the property that is associated with the axiom scheme.

Exercise 6.F

Construct a matrix that indicates which axiom schemes holds in which logic. Let the rows denote the logics listed on page 110, and the columns the axiom schemes. The matrix will then have a total of 36 cells. Then write a '+' or '−' in each cell, according to whether you think the axiom scheme is always true in the logic, or you think that it need not always hold.

It is ultimately the choice of axiom schemes to be included that distinct a particular modal logic. Regardless of which *interpretation* is given to the operators $\Box$ and $\Diamond$, the choice of axiom schemes determines how the symbols may be reasoned with (which, in turn, determines whether a particular interpretation is sensible.)

In this perspective, the most important axiomatic systems of modal logic are:

axiomatic system	axioms included
K	K
D	K + D
T	K + D + T
S4	K + D + T + 4
S5	K + D + T + 4 + 5 + B

As you can see, the K-axiom is included in all of these logics. The scheme forms a basis on which all others are built, and therefore the modal logic **K** is the *weakest* of the modal logics. In literature the list of axioms included for a specific axiom system is usually shorter than the list we presented above. System **S4** is typically characterized by the axioms K, T and 4. And system **S5** is typically characterized by the axioms K, T, 4 and 5. This is due to the fact that some axioms are automatically included if a certain set of other axioms is included. For instance, if axiom T (reflexive) is included, automatically axiom D (serial) holds as well. And if axiom 5 (Euclidian) is included, automatically axiom B (symmetry) holds as well.

6.5 Possible worlds and Kripke semantics

In Section 6.1 we decided whether certain statements were true or not, by intuition. Now, we will formalize this. When we say that 'it is raining' is not necessarily true, we do so because we can imagine a world in which it is not raining. Such a world is called a *possible world.* If in all possible worlds, it is raining, then we must conclude that it is necessarily true for it to rain. This way of deciding the truth of formulas of modal logic, is called the possible world semantics. (In logic, *syntax* defines the rules by which well-formed formulas are constructed, while *semantics* describe the meaning of formulas. Thus, syntax is about shape, and semantics about meaning and truth.)

You may think that in the possible world semantics, the formula $\Box f$ describes the situation that the formula f holds in all worlds, and the formula $\Diamond f$ the situation that the formula f holds in at least one world. But this has undesired consequences. For instance, the formula

$$\Box f \to f$$

would then always be true (because if it holds in *all* worlds, it also holds in our current actual world), whereas we saw in the previous section that this formula is not supposed to hold in general (remember the modal logic of beliefs). So, we have to construct something a bit more complicated.

The necessary additional ingredient, is that we should specify which worlds are 'visible' from within any given world. This means that for every world, we will specify a set of *accessible* worlds, which are 'visible from its vantage point.' The meaning of $\Box f$ will then be that it is true in a world x, exactly in the case that f is true in all *accessible* worlds of x. Similarly, $\Diamond f$ is true in a world x if and only if there is at least one world, accessible to x, in which f is true. This is formalized in Definition 6.8, later on.

To understand why this notion of accessibility is actually quite reasonable, it is useful to consider the case of temporal logic. In a possible world semantics for temporal logic, the worlds accessible to a certain world x would be that world x itself, in addition to all subsequent worlds (in time). Which means that the definition given above of $\Box f$ translates to: 'f is true from this point on.'

A collection of worlds, together with a relation of accessibility, is called a *Kripke model*, named after the American philosopher and logician Saul Kripke. We will now give a clean mathematical definition of these models.

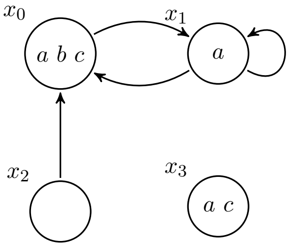
Definition 6.5
A *Kripke model* $\mathcal{M} = \langle W, R, V \rangle$ consists of:

- a non-empty set W of *worlds*
- a function R such that for each world $x \in W$, the set $R(x) \subseteq W$ is the set of *accessible worlds* of x. The set $R(x)$ is also called the *successor* of world x. The function R can also be seen as a relation, which is then called the *accessibility relation.*
- a function V such that for each world $x \in W$, the set $V(x)$ is the set of *atomic propositions* that are true in world x.

We will draw Kripke models as directed graphs, where nodes denote worlds, and arrows between worlds denote accessibility. In the nodes, we write which atomic propositions hold in the corresponding world.

Example 6.6
Here is an example of a Kripke model, which we will name M_1:

M_1 :



In this model there are four worlds $W = \{x_0, x_1, x_2, x_3\}$. The accessibility relation of this Kripke model is given by:

$$\begin{array}{rcl} R(x_0) & = & \{x_1\} \\ R(x_1) & = & \{x_0, x_1\} \\ R(x_2) & = & \{x_0\} \\ R(x_3) & = & \varnothing \end{array}$$

and the worlds make the following atomic propositions true:

$$\begin{array}{rcl} V(x_0) & = & \{a, b, c\} \\ V(x_1) & = & \{a\} \\ V(x_2) & = & \varnothing \\ V(x_3) & = & \{a, c\} \end{array}$$

E.g., the propositions a, b, and c are true in world x_0, but the proposition d is *not* true in world x_0.

We now formalize when formulas are to be held true in a given model $\mathcal{M}$ and world x. First, some notation:

Definition 6.7
The statement "the formula f of modal logic is true in a world x of a Kripke model $\mathcal{M}$" is denoted by

$$\mathcal{M}, x \Vdash f$$

When the model is clear from the context, this is sometimes shortened to:

$$x \Vdash f$$

Typically, $x \Vdash f$ is pronounced as 'world x *satisfies* formula f'. Or simply as 'x satisfies f'. And when $x \Vdash f$ does not hold, we write $x \nVdash f$, with $\Vdash$ struck out, and pronounce it as 'x does not satisfy f'.

The truth of a formula is usually tied to a specific world, in which case it corresponds exactly to the truth definition of propositional logic as we have seen before. The only formulas in which more than one world is taken into consideration are those formed with the modal operators. This observation naturally leads to the following definition of the truth of a given formula f:

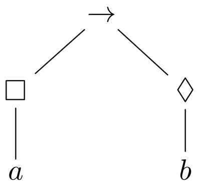
Definition 6.8
Consider a Kripke model $\mathcal{M} = \langle W, R, V\rangle$. Let $x \in W$, p be a propositional atom, and f and

g formulas of modal logic. Then we define:

$$\begin{aligned}
x \Vdash p &\quad \text{iff} \quad p \in V(x) \\
x \Vdash \neg f &\quad \text{iff} \quad x \nVdash f \\
x \Vdash f \wedge g &\quad \text{iff} \quad x \Vdash f \text{ and } x \Vdash g \\
x \Vdash f \vee g &\quad \text{iff} \quad x \Vdash f \text{ or } x \Vdash g \\
x \Vdash f \rightarrow g &\quad \text{iff} \quad \text{if } x \Vdash f \text{ then } x \Vdash g \\
x \Vdash f \leftrightarrow g &\quad \text{iff} \quad (x \Vdash f \text{ iff } x \Vdash g) \\
\\
x \Vdash \Box f &\quad \text{iff} \quad \text{for all } y \in R(x)\text{, it holds that } y \Vdash f \\
x \Vdash \Diamond f &\quad \text{iff} \quad \text{at least one } y \in R(x) \text{ exists, for which } y \Vdash f
\end{aligned}$$

Remark 6.9
The *modal* character of course lies in the last two lines of this definition. The rest are exactly the same as for propositional logic.

Let us now check whether $\Box a \rightarrow \Diamond b$ is true in world x_0 of the Kripke model M_1 we defined above. This formula has the structure:



Hence, we first ask ourselves whether $\Box a$ and $\Diamond b$ are true in x_0. For the first, we have to check *all* successors of x_0, which in this case is just x_1. And indeed, a is true in x_1, that is, $x_1 \Vdash a$, and so $\Box a$ is true in x_0, or, $x_0 \Vdash \Box a$. For $\Diamond b$ to be true in x_0, we must find a successor of x_0 in which b is true. But x_1 is the only successor of x_0, and in fact $x_1 \nVdash b$, so that $x_0 \nVdash \Diamond b$.

Now we turn to the truth table of implication. We know that $\Box a$ is true in x_0, and $\Diamond b$ is not true in x_0. And thus, $\Box a \rightarrow \Diamond b$ is not true in x_0. That is:

$$x_0 \nVdash \Box a \rightarrow \Diamond b$$

Check for yourself in which worlds of M_1 the formula $\Box a$ holds. If you do this correctly, you will find that the formula in fact holds in all worlds of M_1. We denote this by $M_1 \models \Box a$

Definition 6.10
A formula f is said to be *true in a Kripke model* $\mathcal{M}$ if for all worlds x of $\mathcal{M}$ we have $\mathcal{M}, x \Vdash f$. This is denoted by:

$$\mathcal{M} \models f$$

It is pronounced as 'Kripke model $\mathcal{M}$ satisfies formula f', or '$\mathcal{M}$ satisfies f' in short.
Note the distinct meanings of the symbols '$\vDash$' and '$\Vdash$'!

Remark 6.11
Note that in our example, in the world x_2, the formula a is *not* true, that is, $x_2 \nVdash a$. At the same time, we have seen the truth of $M_1 \vDash \Box a$. Hence we see how in a Kripke model, the truth of $\Box f$ does not imply that f is true in all worlds!

Exercise 6.G
For each formula f listed below, check in which worlds x of our Kripke model in Example 6.6 it holds, that is, for which x we have $x \Vdash f$. In addition, check whether f holds in the model, that is, check whether $M_1 \models f$.

(i) a	(iii) $\Diamond a$	(v) $\Box a \to \Box\Box a$
(ii) $\Box a$	(iv) $\Box a \to a$	(vi) $a \to \Box\Diamond a$

Exercise 6.H

(i) Can you find a Kripke model M_2 for which $M_2 \models a$ and at the same time $M_2 \not\models \Box a$? If so, provide such a model; otherwise, explain why such a model cannot exist.

(ii) Provide a Kripke model M_3 for which $M_3 \models a$ although $M_3 \not\models \Diamond a$. Or, if no such model can exist, explain why.

Definition 6.12

Within Kripke semantics, a formula f is said to be *logically true* if for all Kripke models $\mathcal{M}$ we have $\mathcal{M} \vDash f$. This is denoted by:

$$\models f$$

You might want to place restrictions on the structure of Kripke models. A number of such restrictions correspond to the axiom schemes we saw in the previous section. For instance, you might want to impose every world x to have an arrow from and to itself, that is, $x \in R(x)$. A Kripke model that obeys this restriction is called *reflexive*. And the set of reflexive Kripke models indeed corresponds with the axiom scheme T, that is, $\Box f \to f$. Or, you might want to impose that every world has at least one outgoing arrow. This is then called a *serial* Kripke model, and these models correspond to the scheme D, that is, $\Box f \to \Diamond f$. Note how our model M_1 is neither reflexive nor serial. (Why?)

Of the presented axiom schemes, one is *always* true in any Kripke model. This is the axiom scheme K:

$$\Box(f \to g) \to (\Box f \to \Box g)$$

Exercise 6.I

(i) Does a Kripke model M_4 exist that is serial but not reflexive? If so, provide an example, and otherwise, explain why this is not possible.

(ii) Does a Kripke model M_5 exist that is reflexive but not serial? If yes, provide an example, and otherwise, explain why this is not possible.

Using these restrictions we can extend Definition 6.12 to:

Definition 6.13

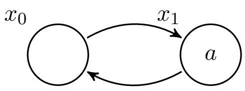
We introduce three new concepts:

- We define '$\models_K f$' as '$\models f$', so as 'f holds in all Kripke models'.
- We define '$\models_T f$' as '$\models f$ holds for all *reflexive* Kripke models'.
- We define '$\models_D f$' as '$\models f$ holds for all *serial* Kripke models'.

Example 6.14

Which (if any) of $\models_K \Box a \to a$, $\models_T \Box a \to a$, or $\models_D \Box a \to a$ hold(s)? As $\Box a \to a$ is an instance of the reflexivity axiom T, $\models_T \Box a \to a$ does hold, as this axiom holds in all reflexive models. However, $\models_D \Box a \to a$ does not hold. Take for instance this model:

M_6 :



As each world has at least one outgoing arrow, it is indeed a serial model. However, $x_0 \Vdash \Box a$ holds, but $x_0 \Vdash a$ does not hold. Therefore, $x_0 \Vdash \Box a \to a$ also does not hold and hence $M_6 \vDash \Box a \to a$ does not hold. And hence $\models_D \Box a \to a$ does not hold. By the way, does this model resembles your model M_4 in Exercise 6.I a bit?

And obviously, if $\models_D \Box a \to a$ does not hold, then $\models_K \Box a \to a$ also does not hold as we can simply take the same counter example M_6 again.

6.6 Temporal logic

We will now take a more detailed look at a specific temporal logic: LTL, which stands for *Linear Time Logic*. This is the logic that the SPIN model checker uses.[1] A *model checker* is a computer program that analyzes pieces of software. In order to use a model checker, one first has to provide a finite automaton that describes the behavior of the software in question. Then, that automaton is fed into the checker, together with a formula of the temporal logic employed by the checker that expresses the property one wants to check of the software. The model checker then analyzes the automaton and verifies whether the property expressed by the formula holds or not.

An example of a property one may want to give to a model checker would be:

The automaton always eventually returns back to the state q_{42}.

(A property of this form is called a 'liveness' property, because it describes how the automaton always remains 'active' enough to be able to return back to a certain state.) A temporal logic formula that expresses this property would be:

$$\Box\Diamond q_{42}$$

This should be read as:

From now on it will always hold that, there will be a point in time at which, we are back in state q_{42}.

You might be tempted to think the model checker's task is not very complex, and thus actually quite simple. The complication however, is that, even with automata of a relatively small number of states, the number of situations (worlds) that need to be checked grows very (very) fast. This, then, is the key task of a model checker: controlling what is called *state space explosion.*

Definition 6.15
In LTL, we don't denote the modal operators by $\Box$ and $\Diamond$, but by the calligraphic capitals $\mathcal{G}$ and $\mathcal{F}$. So we have:

modal	LTL	shorthand	meaning
$\Box f$	$\mathcal{G}\,f$	$\mathcal{G}$lobally	from now on (including now) f will always hold
$\Diamond f$	$\mathcal{F}\,f$	$\mathcal{F}$uture[2]	eventually f will hold (or it holds already)

In fact, LTL has even more operators:

LTL	shorthand	meaning
$\mathcal{X}\,f$	ne$\mathcal{X}$t	after exactly one step, f will hold
$f\;\mathcal{U}\;g$	$\mathcal{U}$ntil	f holds until g holds, and indeed eventually g will hold
$f\;\mathcal{W}\;g$	$\mathcal{W}$eak until	f holds until g holds, or else f will hold forever
$f\;\mathcal{R}\;g$	$\mathcal{R}$elease	g holds until f holds, at which point g is still the case, or else g will hold forever

As you can see, LTL has binary modal operators as well.

[1] SPIN stands for 'Simple Promela INterpreter' and Promela in turn stands for 'PRocess MEta LAnguage', which allows the description of finite automata. SPIN is one of the most well-known model checkers, and was developed by the Dutch Gerard Holzmann at Bell Laboratories in America.

[2] Note that there are also textbooks that state that $\mathcal{F}$ stands for 'Finally' instead of 'Future'. However, as this chapter is based on the book [2] we follow its definition and refer to 'Future'.

This is a grammar for LTL, where a represents any propositional variable.

$$\begin{array}{rcl} S & \rightarrow & a \mid \neg S \mid (S \wedge S) \mid (S \vee S) \mid (S \rightarrow S) \mid (S \leftrightarrow S) \mid \\ & & \mathcal{G}\, S \mid \mathcal{F}\, S \mid \mathcal{X}\, S \mid (f\; \mathcal{U}\; g) \mid (f\; \mathcal{W}\; g) \mid (f\; \mathcal{R}\; g) \end{array}$$

As always, the unary operators bind the strongest. For the binary LTL operators, we do not define a specific binding strength, which means that we have to write parentheses as in the grammar to parse formulas in a unique way. And note that within literature the operator $\mathcal{X}$ is sometimes also denoted with a small circle as $\circ$, but not in this course, so don't use it!

Exercise 6.J
Express the following sentences in LTL. You may use all operators of LTL in your answers.

(i) *There exists a moment after which the formula a will always be true.*
(ii) *The statements a and b are alternatingly true.*
(iii) *Every time a holds, b holds after a while as well.*

Some of the operators of LTL can in fact be expressed in terms of the others. For instance, we have:

$$\begin{array}{rcl} f\; \mathcal{U}\; g & \text{is equivalent to} & (f\; \mathcal{W}\; g) \wedge \mathcal{F}\, g \\ f\; \mathcal{W}\; g & \text{is equivalent to} & (f\; \mathcal{U}\; g) \vee \mathcal{G}\, f \\ f\; \mathcal{R}\; g & \text{is equivalent to} & \neg(\neg f\; \mathcal{U}\; \neg g) \end{array}$$

Remark 6.16
This last example is again an example of a conjugation. But there are more conjugations in LTL, for instance $\neg\, \mathcal{G}\, f \equiv \mathcal{F}\, \neg f$, $\neg\, \mathcal{F}\, f \equiv \mathcal{G}\, \neg f$, $\neg(f\; \mathcal{U}\; g) \equiv \neg f\; \mathcal{R}\; \neg g$, $\neg(f\; \mathcal{R}\; g) \equiv \neg f\; \mathcal{U}\; \neg g$, and $\neg\, \mathcal{X}\, f \equiv \mathcal{X}\, \neg f$. So $\mathcal{X}$ is self-conjugated.

Exercise 6.K
Define the operator $\mathcal{U}$ in terms of $\mathcal{R}$.

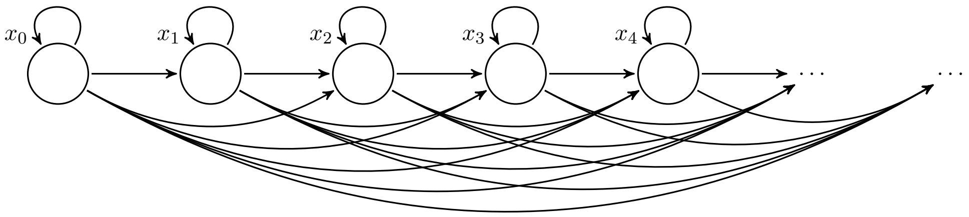
Exercise 6.L
Define the operators $\mathcal{G}$ and $\mathcal{F}$ in terms of $\mathcal{U}$ and $\mathcal{R}$. You may use the propositions $\top$ and $\bot$, which are always true, respectively always false.

Exercise 6.M
Show that all LTL operators (except $\mathcal{X}$) can be defined in terms of the $\mathcal{W}$ operator. Again, you may use the propositions $\top$ and $\bot$.

The worlds in Kripke models of LTL corresponds to the ticking of a clock. Another way to putting it, is that each world can be uniquely identified with a natural number. So, the set of worlds is:

$$W = \{x_i \mid i \in \mathbb{N}\}$$

Furthermore, the worlds accessible to a given world x_i are exactly all subsequent worlds x_j, for which $i \leq j$. The only difference then, between different Kripke models of LTL, is which atomic propositions hold in its worlds. Here is a picture illustrating what a Kripke model of LTL looks like:



Note that this diagram shows that the accessibility relation in the model is transitive, but it is not Euclidean.

This formulation of worlds in terms of the ticking of a clock allows us to describe the meaning of the operators of LTL of Definition 6.15 in an even more mathematically precise way:

Definition 6.17
Given a Kripke model $\langle W, R, V\rangle$ with W and R as described above. Then:

$x_i \Vdash \mathcal{G}\, f$	for all $j \geq i$ we have $x_j \Vdash f$
$x_i \Vdash \mathcal{F}\, f$	there is a $j \geq i$ such that $x_j \Vdash f$
$x_i \Vdash \mathcal{X}\, f$	$x_{i+1} \Vdash f$
$x_i \Vdash f\, \mathcal{U}\, g$	there is a $j \geq i$ such that $x_j \Vdash g$ and for all $k \in \{i, i+1, \ldots, j-1\}$ we have $x_k \Vdash f$
$x_i \Vdash f\, \mathcal{W}\, g$	*either* there is a $j \geq i$ such that $x_j \Vdash g$ and for all $k \in \{i, i+1, \ldots, j-1\}$ we have $x_k \Vdash f$, *or* for all $l \geq i$ we have $x_l \Vdash f$
$x_i \Vdash f\, \mathcal{R}\, g$	*either* there is a $j \geq i$ such that $x_j \Vdash f$ and for all $k \in \{i, i+1, \ldots, j-1, j\}$ we have $x_k \Vdash g$, *or* for all $l \geq i$ we have $x_l \Vdash g$

With this formalization, we could prove the above-mentioned equivalences between LTL operators.

Exercise 6.N
Consider the LTL Kripke model $M_7 = \langle W, R, V\rangle$. So we know that $W = \{x_i \mid i \in \mathbb{N}\}$ and $x_j \in R(x_i)$ if $i \leq j$. Now define V as follows:

$$\begin{aligned} a \in V(x_i) \quad &\text{iff} \quad i \text{ is a multiple of two} \\ b \in V(x_i) \quad &\text{iff} \quad i \text{ is a multiple of three} \end{aligned}$$

Check whether the following properties hold:
(i) $x_0 \Vdash \mathcal{F}(a \wedge b)$
(ii) $x_6 \Vdash \mathcal{G}(a \vee b)$
(iii) $M_7 \vDash \mathcal{G}\mathcal{F}(a\, \mathcal{U}\, b)$

Exercise 6.O
Which of the axiom schemes in the table on page 112 hold in LTL?

6.7 Important concepts

Appendix A

Preliminaries on set notations

This appendix contains information about sets that we hope you have already seen before. If not, please read this section on your own and ask questions about it during the tutorial. The contents will probably not be explained during the plenary lectures.

In Chapter 2 we already talked a lot about sets (the set of all women, the set of the integers, et cetera), but we didn't really explain how sets are typically represented.

Basically, a set is just a collection of unordered objects.

- If the amount of objects is finite, we say that the set is finite, and the typical way to represent such a finite set is by *enumerating* all elements between curly braces and separated by commas:

  $$\{a, b, c, d, e, f, g, h, i, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$$

  Although technically feasible, for very large (but still finite) sets, this is a lot of work and it is very easy to forget elements (like the j above).

- But for infinite sets this obviously doesn't work at all, because we are never able to enumerate a complete infinite list of objects. If the structure of the sets is really clear, we may use some 'ellipses' or '. . . ' to visualize that we left out some elements, but that you should be able to determine anyway whether a given object is indeed a member of the set.

  - For instance $\{a, b, c, \ldots, z\}$ should indicate that this set contains all lower case letters from the English alphabet, including the j. But if we specifically want to leave out this j, we could have written something like $\{a, b, c, \ldots, h, i, k, l, \ldots, z\}$.
  - And the infinite set of natural numbers that are greater than or equal to seven can be represented as $\{7, 8, 9, \ldots\}$.
  - And the infinite set of the integers can be represented using two ellipses as in $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, but also using only one, as in $\{0, -1, 1, -2, 2, -3, 3, \ldots\}$, because the pattern is obvious.

- However, there is a very powerful representation system called *set-builder notation* or *set comprehension*, which is the main method being used to represent sets.

- Its basic form is

  $$\{x \mid P(x)\}$$

  where the x before the vertical bar is a variable, the bar itself is just a separator, and the $P(x)$ behind the bar is a predicate on x. Or, slightly shorter, this is the set of all x such that $P(x)$ holds.

- It should be read as: this set contains all elements x for which the predicate $P(x)$ holds.
- Very often the variable x is restricted to a certain domain D, which can be represented in two ways:
  $$\{x \in D \mid P(x)\} \quad \text{or} \quad \{x \mid x \in D \text{ and } P(x)\}$$
  Both are representations of the set that contains all elements x from domain D for which predicate $P(x)$ holds.
- Often when it is clear about which domain we are talking, this explicit domain is omitted.
- So if we would like to represent the set of all positive real numbers, we could do so by first stating that $P(x)$ is true if x is a positive real number, and then giving this definition: $\{x \in \mathbb{R} \mid P(x)\}$. However, for this kind of simple mathematical predicates we usually use the existing mathematical notation for it, so we would get
  $$\{x \in \mathbb{R} \mid x > 0\}$$
- Note that it is not obligatory to express this predicate in mathematical symbols. It can also be expressed using natural language:
  $$\{w \in \{a, b\}^* \mid w \text{ contains an even number of } a\text{'s}\}$$
- More examples:
  - $\{x \in \mathbb{Q} \mid x \leq 0\}$ is the set of all rational numbers that are not positive.
  - $\{x \in \mathbb{Z} \mid |x| = 1\}$ is simply the finite set $\{-1, 1\}$.
  - $\{z \mid \exists k \in \mathbb{Z}\,[z = 2k]\}$ is the set of all even integers.
- So far we have only seen sets where there is only a single variable (possibly with a limiting domain) before the vertical bar, but if we want to give more *structure* to the elements of the set, we may put a *term* at that position that contains more than one variable.
- More examples:
  - $\{2z \mid z \in \mathbb{Z}\}$ is another representation for the set of all even integers.
  - $\{(n + 1, 2n + 1) \in \mathbb{N} \times \mathbb{N} \mid n \in \mathbb{N}\}$ is the set that contains ordered tuples of natural numbers where the first number is some kind of index and the second number is the corresponding odd natural number. In other words we get the set $\{(1, 1), (2, 3), (3, 5), (4, 7), \ldots\}$.
  - $\left\{a^n b^{2n} \in \{a, b\}^* \mid n \in \mathbb{N}\right\}$ is the set of words over the alphabet $\{a, b\}$ where each word starts with a certain amount of a's and is followed by exactly the double amount of b's.
- Note that in the last two examples before the bar an explicit domain is given. However, because this domain directly follows from the fact that $n \in \mathbb{N}$ as is given behind the bar, we may omit the explicit domains without any problems and we get $\{(n + 1, 2n + 1) \mid n \in \mathbb{N}\}$ and $\left\{a^n b^{2n} \mid n \in \mathbb{N}\right\}$.

Bibliography

[1] W. Gielen. Discrete wiskunde voor informatici. Katholieke Universiteit Nijmegen - Subfaculteit Wiskunde 2002.

[2] M. Huth and M. Ryan. *Logic in Computer Science.* Cambridge University Press, 2004. ISBN 0-521-54310-X.

[3] T.A. Sudkamp. *Languages and Machines.* Pearson Education Inc., third edition, 2006. ISBN 0-321-31534-0.

[4] J.F.A.K. van Benthem, H.P. van Ditmarsch, J.S. Lodder, J. Ketting, and W.P.M. Meyer-Viol. *Logica voor informatici.* Pearson Addison-Wesley, Nederland, 2003. ISBN 90-430-0722-6 (dit boek wordt ook gebruikt in het college Beweren en Bewijzen).

[5] R.J. Wilson. *Introduction to Graph Theory.* Pearson Education Limited, fifth edition, 2010. ISBN 978-0-273-72889.

Index