COMP11120 Mathematical Techniques for Computer Science
Semester 2

Andrea Schalk
A.Schalk@manchester.ac.uk

Course webpage
studentnet.cs.manchester.ac.uk/ugt/COMP11120

April 3, 2018
Chapter 6

Recursion and Induction

Recursion is a powerful principle. It can be used to

• define sets,
• define data types in some programming language such as Java, C, and Haskell,
• define operations on recursively defined sets,
• define operations on recursively defined datatypes and
• design recursive algorithms.

Recursion as a tool for defining data types and operations on these is covered in COMP11212 and as a tool in designing algorithms in COMP16212 and COMP26120. Recursion goes hand in hand with a proof principle which is known as induction.

There is a general principle at work in all these.

• In order to recursively define an entity we need
  – a number of base cases—these create particularly simple instances of our set or data type and
  – a number of step cases—these create more complicated instances from simpler ones already available.

• In order to define an operation on such a recursively defined entity all one has to do is
  – define what the operation should do for each of the base cases and
  – define what the operation should do for each step case, assuming that it has already been defined for the simpler entities.

• In order to prove that a recursively defined entity or operation has a particular property all one has to do is
  – check each base case and
  – assuming that the property holds for simpler constructs (known as the induction hypothesis) show that it holds for after applying each step case.
Some of you will have come across natural induction as a proof principle. This is a special case of induction which relies on the idea that the natural numbers can be constructed via recursion:

- the base case is given by the number 0 and
- the step case is given by the fact that every number \( n \) has a unique successor, the number \( n + 1 \).

We illustrate the general idea with a number of examples. We begin with the idea of recursive data types such as lists and trees. Many programming languages allow us to define recursive data types, typically using pointers or references. You will meet these ideas in COMP26120 (pointers in C) and COMP16212 (trees in Java).

When computer scientists define syntax they often use recursive definitions. For example, regular expressions (Example 6.24) are defined recursively, and in any programming language the definition of what constitutes a valid program provides another example of a recursive definition (see Example 6.30), typically using something called a grammar, see Examples 6.27 and 6.28.

These topics are taught in COMP11212, but the recursive nature of the definitions is not the main focus, so we briefly discuss that here.

We finally look at more mathematical examples, based on the natural numbers. A recurring theme on this course unit has been the fact that mathematics is all about rigour. There are a number of formal definitions in the notes in preceding chapters, while we have taken for granted certain facts about sets of numbers, and familiar operations on these. In this and the following chapter we give an idea of how these sets of numbers, and their operations, can be formally defined. This allows us to formally prove properties for these operations, such as commutativity or associativity, that we have taken for granted. This illustrates that rigour in mathematics is not something built on top of systems of numbers, but instead these can be formally defined, and we don’t have to rely on ‘common sense’ or ‘experience’ to justify properties of their operations.

6.1 Lists

We begin our study of recursion by looking at lists. Lists are a standard recursive data type. The idea is quite simple: For a set \( S \) we can have a list of elements from \( S \), say

\[ [s_n, s_{n-1}, \ldots, s_2, s_1] \]

For example, this might be a list of pages relevant to a particular topic, or a list of heights of people that somebody has measured, or a list of students on a particular course unit.

6.1.1 Lists defined

We can think of a list as satisfying exactly one of the following conditions:

- The list is empty, that is, there are no elements in the list.\(^1\)

\(^1\)For example, if we have a program that calculates which elements should be added to the list we would like it to start from an empty list.
The list consists of at least one element, and we can think of it as being a smaller list, say \( l \), together with an added element, say \( s \).

The list
\[
[4, 3, 2, 1]
\]
is the result of appending\(^2\) the number
\[
4
\]
to the list
\[
[3, 2, 1],
\]
which in turn is the result of appending the number
\[
3
\]
to the list
\[
[2, 1],
\]
which in turn is the result of appending\(^3\) the number
\[
2
\]
to the list
\[
[1],
\]
which in turn is the result of appending the number
\[
1
\]
to the list
\[
[].
\]

We can use this idea to give a formal definition of lists over a set \( S \), and we can use the same idea to define operations on such lists.

**Definition 46: list**

A list over a set \( S \) is recursively defined as follows.

**Base case** list. There is an empty list \([\ ]\).

**Step case** list. Given a list \( l \) over \( S \) and an element \( s \) of \( S \) there is a list \( s : l \) where \( s \) has been appended to \( l \).

We use
\[
\text{Lists}_S
\]
for the set of all lists over a given set \( S \).

**Example 6.1.** What is written above regarding the list
\[
[4, 3, 2, 1]
\]
can now be written in the notation that is introduced in the formal definition of a list to read
\[
\]

\(^2\)We use terminology from the language Python for our lists. In ML, the *append* operation concatenates two lists, and the *cons* operation adds an element to the list.

\(^3\)We have made a somewhat arbitrary decision here to append elements to the left of the list. Instead we could have used \([1, 2, 3]\) to mean the list which arises from appending 1, 2 and 3 (in that order) to the empty list.
Always having to deal with expressions like the one in the last row would be quite painful, but that is how a computer thinks of a list which is given by a list element and a pointer to the remainder of the list. Human readability is improved, however, by using expressions like that in the top row. These kinds of recursive definitions are very typical for functional programming languages such as Haskell or ML, but when programming with lists in C you will find a similar idea: An object of class List

- is empty or
- it consists of
  - an element of the list and
  - a pointer to the remainder of the list (if there is one).

**Code Example 6.1.** Here’s a class that implements this kind of list in Java, where the elements of the list are integers.

```java
public class List {
    public int value;
    public List next;

    public List (int s, List l) {
        value = s; next = l;
    }
}
```

We have to cheat slightly to deal with empty lists: We do this by using the value `null` to indicate that an empty list is being referenced. So an object of the class `List` is

- the empty list if we get the value `null` when referring to it or
- it consists of an element `value` and a `List` object `next`.

The fact that the class `List` is closely related to the lists defined in Definition 46 is not immediately obvious, but hopefully this explanation shows why they correspond to each other.

We give an idea of how to picture an object in this class. Assume that we have

- a `List` object `l` with `l.value=4` and `l.next=l3` and
- a `List` object `l3` with `l3.value=3` and `l3.next=l2` and
- a `List` object `l2` with `l2.value=2` and `l3.next=l1` and
- a `List` object `l1` with `l1.value=1` and `l1.next=null`.

You can picture these objects as follows:
Another picture that is sometimes used in this situation is the following.

6.1.2 Recursive definitions and proof by induction

Operations on such recursive datatypes are usually defined recursively. We give a number of examples for this particular construct.

**Example 6.2.** We can define a very simple-minded function from \( \text{Lists}_S \) to \( \text{Lists}_S \), which maps a given list \( l \) over \( S \) to the empty list. Because this function is very simple we do not need recursion to define it, we could merely set

\[
\begin{align*}
\text{Base case } k &: \, k[] &= []
\end{align*}
\]

But the point of this example is to introduce recursive definitions, and so we show here how to define the same function recursively. For this purpose we have to give it a name, say \( k_\perp \) since it is the constant function which maps everything to the empty list. Note that the following definition is inefficient, and one would not use it to program this function, but it is useful as a first simple example for how recursive functions work.

**Base case** \( k_\perp \). \( k_\perp[] = [] \) and

**Step case** \( k_\perp \). \( k_\perp(s : l) = k_\perp l \),

Note that this definition matches the definition of a list: We have to say what the function does if its argument is the base case list, that is, the empty list \( [] \), and we have to say what it does if its argument is a list built using the step case, so it is of the form \( s : l \) for a list \( l \).

The way this function works is to map

- the empty list to the empty list,

- and a non-empty list, which has an element \( s \) added to some list \( l \), to the result of applying the function to \( l \).
This is the typical shape of a recursive function on a recursive data type:

- it specifies what to do for the base case(s) of the data type and
- it specifies what to do for the step case(s).

**Example 6.3.** We continue Example 6.2 by carrying out a sample calculation to see how this definition allows us to compute the value of the function for a specific list, say $[i, 1 + 2i, 4]$ (a list over $\mathbb{C}$). We justify each step by referring to the definition of a list, Definition 46, and the definition of $k[i]$.

$$
k[i][i, 1 + 2i, 4] = k[i](i : [1 + 2i, 4]) \quad \text{step case list}
= k[i][1 + 2i, 4] \quad \text{step case } k[i]
= k[i][1 + 2i] \quad \text{step case list}
= k[i][4] \quad \text{step case } k[i]
= k[i][4 : []] \quad \text{step case list}
= k[i][] \quad \text{step case } k[i]
= [] \quad \text{base case } k[i].
$$

Note that in a typical implementation it would not be necessary to invoke *step case list*—it is our notation for lists which requires this.

**Code Example 6.2.** A code snippet that implements this function as a method `knoll` for an object `l` of class `List` looks as follows:

```java
public static List knull (List l)
{
    if (l == null)
        return null;
    else
        return knull(l.next);
}
```

The way a computer carries out the corresponding calculation looks a bit different to the sample calculation given above: Instead of manipulating an expression that describes the output a computer stores each call to the recursively defined method, and that requires it to also store all the local variables.

**Code Example 6.3.** We look at the function calls and returns in an example. Assume we are calling `knoll(l)`, where `l` is the list from Code Example 6.1. The calls carried out by the program are as follows:

`knoll(l)`
This looks a bit boring but becomes more interesting if the function that is being implemented is more interesting, see Example 6.6.

If we put the mathematical definition of the function next to the implementation the similarities are very clear:

```java
public static List knull (List l)
{
    if (l == null)
        return null;
    else
        return knull(l.next);
}
```

It is not completely obvious from the definition that the function $k[l]$ as given there does indeed map every list to the empty list. We prove this formally as our first example of a proof by *induction*.

We want to show that for all lists $l$ in Lists it is the case that $k[l] = []$.

Such a proof also follows the formal definition of the underlying data type. The pattern consists of a proof for the base case, a proof of the step case, and (optionally here) in between the statement of the *induction hypothesis*.

**Base case** list. We have to show that the statement holds for the empty list, that is $k[[]] = []$.

**Induction hypothesis.** We assume the statement holds for the list$^4$ $l$, that is we have $k[l] = []$.

**Step case** list. We have to show that given the induction hypothesis the statement holds for lists of the form $s : l$, that is we have $k[s : l] = []$.

$^4$Note that sometimes we have to assume that the statement holds for all lists of a given length, or some other statement applying to more than one list.
Example 6.4. We continue with Example 6.2 and illustrate the formal proof of the statement from above.

**Base case** list. \( k_{[]} = [] \). This is a simple application of the base case of the definition of the function \( k_{[]} \).

**Induction hypothesis.** For the list \( l \) we have

\[
k_{[]}l = []
\]

**Step case** list. We check that

\[
k_{[]} (s : l) = k_{[]} l \quad \text{step case} \quad k_{[]} = [] \quad \text{induction hypothesis.}
\]

As is typical the base case is obvious, and it typically does not require many steps, while the step case has a little more substance.

Note that we were able to define this function without having to specify from which set \( S \) our lists take their elements.

Let’s pause a moment to think about how this proof works. The base case is fairly easy to understand. It seems as if in the middle of the proof, where the induction hypothesis is stated, we are assuming the very same thing we aim to prove. This is not so, however. An analogy that is often invoked is that of a line of dominoes. Assume that you have got a line of dominoes, standing on their short side, starting in front of you, and extending to the right (you may imagine infinitely many dominoes).

**Base case.** The first domino falls over, to the right.

**Induction hypothesis** The \( n \)th domino falls over to the right.

**Step case.** If the \( n \)th domino falls over to the right then the \( (n + 1) \)th domino falls over to the right.

In this way of writing the proof the induction hypothesis is often skipped because it appears in the step case (‘If the \( n \)th domino falls over . . . ’). If we do not state it explicitly we do not lose any information. After the first few examples I do not note the induction hypothesis explicitly if it is a precise copy of the statement we are proving; See Section 6.4 for examples where more sophisticated induction hypotheses appear.

The base and the step case together are sufficient to guarantee that if the first domino falls over to the right (the base case) then all dominoes fall over.

Another analogy you may find helpful is that of climbing a ladder.

**Base case.** I can climb the first rung of the ladder.

**Step case.** If I can get to the \( n \)th rung of the ladder then I can climb to the \( (n + 1) \)th one.
If we have both these properties, then we can get onto the first, and all subsequent, rungs of the ladder.

Going back to the case of lists, the base case ensures that we know the desired result for the empty list, and the step case ensures that if we know the result for the list \( l \) then we can prove it for the list where some element \( s \) had been added to \( l \) to form \( s : l \).

**Example 6.5.** We look at the proof that \( k[l] = [] \) for a specific list \( l \).

Let

\[
l = [s_3, s_2, s_1] = s_3 : (s_2 : (s_1 : [])).
\]

We can see that in order to show that

\[
k(s_3 : (s_2 : (s_1 : []))) = []
\]

we need the step case three times:

\[
k(s_3 : (s_2 : (s_1 : []))) = k(s_2 : (s_1 : [])) \quad \text{(step case \( k[l] \))}
\]

\[
= k(s_1 : []) \quad \text{(step case \( k[l] \))}
\]

\[
= [] \quad \text{(base case \( k[l] \))}
\]

We can also see that if we have a list with five elements then we need the step case five times. A proof by induction takes a shortcut: by proving the base case, and that given a list with \( n \) elements which has the desired property we can show that adding another element to the list results in a list that also has the property in question, we can establish the property for all finite lists.

Note that sometimes in order to establish the step case (here \( s : l \)) we have to assume that the induction hypothesis for all entities which are somehow smaller than the current one. In the case of lists it might be that we assume the induction hypothesis for all lists we have built on the way to reach the list \( l \), or possibly even for all lists which have at most as many elements as \( l \).

A proof by induction for lists always takes the following shape:

**Base case** list. The property is established for the base case for lists, that is: We show it works for \([\ ]\). If the property we want to show is an equality then we get the statement we want to prove by inserting the empty list \([\ ]\) for every occurrence of \( l \) in the equality, so for example

\[
k[l] = []
\]

becomes

\[
k[] = [],
\]

and\(^5\)

\[
\text{sum} \ rev \ l = \text{sum} \ l
\]

\(^5\)Read on to find out how the operations used here are defined, but that is not required for understanding the point that is made here.
becomes

\[ \text{sum rev } [] = \text{sum } []. \]

**Ind hypothesis.** We assume that the statement holds for the list\(^6\) \(l\).

**Step case** list. Assuming the induction hypothesis we show that the property holds for the list \(s : l\). We obtain the statement we have to show by replacing every occurrence of \(l\) by \(s : l\), so

\[ k_{[]} l = [] \]

becomes

\[ k_{[]} (s : l) = [], \]

and

\[ \text{sum } \text{rev } l = \text{sum } l \]

becomes

\[ \text{sum } \text{rev}(s : l) = \text{sum}(s : l). \]

We summarize these ideas as follows.

<table>
<thead>
<tr>
<th><strong>Tip</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A proof by induction for lists over a given set (S) always has the following shape. Assume we are trying to show a statement given for all (l \in \text{Lists}_S).</td>
</tr>
</tbody>
</table>

**Base case** list. Prove the given statement for the case where all occurrences of \(l\) have been replaced by \([]\).  

**Ind hyp** Assume the given statement holds for the list \(l\).\(^7\)

**Step case** list. Prove the statement where all occurrences of \(l\) have been replaced by \(s : l\), where \(s\) is an arbitrary element of \(S\). The induction hypothesis is used as part of the proof.

Note that when proving something by induction for a recursively defined structure other than the natural numbers, people often speak of *structural induction*, because the structure of the data type gives the shape of the induction argument.

### 6.1.3 Operations on lists

What more sophisticated operations are available on lists? Well, for example one might add up all the numbers in a list.

**Example 6.6.** The following is part of a previous exam question. Assume that \(N\) is a set of numbers \(\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}\) or \(\mathbb{C}\). We want to define a

\(^{6}\)Note that for advanced properties one may have to extend this, and, for example, assume the statement for all lists built on the way to reaching \(l\), or that it holds for all lists that have at most the same number of elements as \(l\), but on this course we have no examples like that.

\(^{7}\)It may be necessary to assume it for \(l\) and all sublists of \(l\).
function

\[ \text{sum}: \text{Lists}_N \rightarrow N \]

which adds up all the members of a given list. In other words we would
like to have

\[ \text{sum}[4, 3, 2, 1] = 10. \]

**Tip**
Before defining an operation that is described verbally it can be a good
idea to think about what that operation should do to a concrete example
to make sure that you understand the instructions given.

**Example 6.7.** Continuing our example we ask ourselves how do we
define such an operation recursively? We have to ask ourselves: What
do we want to happen in the base case? So what should it mean to add
up all the numbers that appear in the empty list? The answer is that
we should get the unit\(^8\) for addition, 0.

We then have to think about what we want to happen in the step case,
so knowing the result of \(\text{sum}\ l\), what should the result of \(\text{sum}(s : l)\) be?
It can sometimes help to write it like that, namely

\[ \text{sum}(s : l) = ??? \text{ sum } l. \]

Well, if we already know what happens if we add up all the numbers
of the list \(l\), then to add up all the numbers in the list \(s : l\), we just have to
add \(s\) to the result. These considerations lead to the following definition.

**Base case** \(\text{sum}\). \(\text{sum}[\ ] = 0.\)

**Step case** \(\text{sum}\). \(\text{sum}(s : l) = s + \text{sum } l.\)

Note how this definition very closely follows the shape of the definition
of a list. This is one of the hallmarks of definition by recursion. If you
understand the definition of the underlying entity you know what shape
an operation for the type will have. A sample calculation (this time
without using our notation for lists to stay closer to the programming
example below) looks as follows:

\[
\begin{align*}
\text{sum}(4 : (3 : (2 : (1 : [])))) \\
&= 4 + \text{sum}(3 : (2 : (1 : [])))) \quad \text{step case sum} \\
&= 4 + 3 + \text{sum}(2 : (1 : [])) \quad \text{step case sum} \\
&= 4 + 3 + 2 + \text{sum}(1 : []) \quad \text{step case sum} \\
&= 4 + 3 + 2 + 1 + \text{sum}[] \quad \text{step case sum} \\
&= 4 + 3 + 2 + 1 + 0.
\end{align*}
\]

See Example 6.11 for an inductive proof of a property of this function.

\(^8\)If you were wondering how to deal with the empty list then note that adding no
elements at all is usually taken to describe the number 0. More generally, for an operation
with unit \(e\), applying the operation to 0 many elements should return \(e\).
Tip
When you have defined an operation recursively it is always a good idea to apply it to an example to make sure that your definition makes sense. Note how this is done in the preceding example.

Code Example 6.4. For our class List this method could be implemented by the following method add.

```java
public static int sum (List l)
{
    if (l == null)
        return 0;
    else
        return l.value + sum(l.next);
}
```

We can work out the function calls for the calculation for the lists $l$, $l_3$, $l_2$, $l_1$ as in Code Example 6.1, as follows:

```
sum(l)
  sum(l3)
    sum(l2)
      sum(l1)
        sum(null)
          return 0
          return 1 + 0
          return 2 + 1
          return 3 + 3
          return 4 + 6
```

You can see why the computer has to create a stack which contains all the information needed to know what to do with the various return values and where to continue the computation.

Again note the similarities between the mathematical function definition and the code implementing it.
In general, defining a new operation for lists always takes the following shape:

**Base case** We have to say what the operation should do when given the base case of a list, that is, [].

**Step case.** Assuming that we already know what the operation does for the list \( l \) we define what it does when the argument is \( s : l \).

We define an operation

\[
\text{Lists}_S \times \text{Lists}_S \rightarrow \text{Lists}_S
\]

for a more interesting example.

**Example 6.8.** We want to define an operation that takes as input two lists, and returns one list where the two have been stuck together, that is, *concatenated*, so

\[
[1, 2, 3] + [4, 3, 2, 1] = [1, 2, 3, 4, 3, 2, 1].
\]

We would like to have a formal definition of this operation. This formal definition can then be turned into code that implements this operation. Now that we have two arguments we need to think about how to work that into the definition. For simple operations such as this one it is possible to consider one argument as a *parameter* and give a recursive definition in terms of the other argument. In this case we can leave the right argument alone and only have to look inside the left one, which is why we recurse over the *left* argument.

We define the operation

\[
+: \text{Lists}_S \times \text{Lists}_S \rightarrow \text{Lists}_S
\]

by setting

**Base case** 
\([] + l' = l'\) and

**Step case** 
\((s : l) + l' = s : (l + l')\),

where it is understood that \( l \) and \( l' \) are elements of \( \text{Lists}_S \).

Note that the only way to define this is by recursion over the left argument. If we try to do it the other way round we have no way of defining the list we want from

\[ l + (s : l'). \]
In Exercise 112 we look at a way of doing this recursing over the right argument, but that requires another operation and is considerably more complicated.

**Example 6.9.** We continue Example 6.8 by illustrating how this operation works in practice. We show how two lists are concatenated step by step by following the process defined by this definition.

\[
[4, 3] + [2, 1] = (4 : [3]) + [2, 1] \quad \text{step case list}
\]

\[
= 4 : ([3] + [2, 1]) \quad \text{step case } +
\]

\[
= 4 : ((3 : []) + [2, 1]) \quad \text{step case list}
\]

\[
= 4 : 3 : (([] + [2, 1]) \quad \text{step case } +
\]

\[
= 4 : 3 : [2, 1] \quad \text{base case } +
\]

\[
= 4 : [3, 2, 1]
\]

\[
= [4, 3, 2, 1].
\]

**Code Example 6.5.** We give the code that corresponds to the mathematical definition.

```java
public static List concat (List l, List l2)
{
    if (l == null)
        return l2;
    else
        return new List(l.value, concat(l.next,l2));
}
```

The code also illustrates why we have to use recursion over the left argument (try writing code which does it over the right argument if you can’t see why—you just don’t have access to the relevant part of the list).

From the point of view of programming we usually stop at defining operations, but when writing a compiler we have to worry about the properties of such operations.

We illustrate here how the proof principle of induction allows us to establish properties for operations that have been defined using the principle of recursion.

Such proofs once again follow the shape of the original definition of the operation, which in turn follows the shape of the definition of the original entity.

**Example 6.10.** We show that the empty list works like a unit on the right, that for all lists \( l \) over a set \( S \), and all \( s \in S \), we have

\[
l + [s] = l.
\]

We follow the shape of the definition of the concatenation operation, by
first considering the base case.

**Base case** list. We note that

\[ \[ \] + \[ \] = \[ \] \quad \text{base case } \oplus . \]

**ind hyp** We assume that the statement holds for the list \( l \), that is

\[ l + \[ \] = l. \]

We turn to the step case.

**Step case** list. We calculate

\[ (s : l) + \[ \] = s : (l + \[ \]) \quad \text{step case } \oplus \]

\[ = s : l \quad \text{induction hypothesis}. \]

Taken together these two cases cover all possibilities, and they form a proof by induction.

Note that we have immediately from the base case of the definition of \( \oplus \) that \( \[ \] \) acts like a unit on the left, so overall we can see that it is the unit for this operation.

Below on we do not write the induction hypothesis explicitly if it consists of a statement that is identical to the one we are proving. In other words in examples of induction proofs which do not have an induction hypothesis spelled out you may assume that it looks like the statement we are proving.

**CExercise 109.** Answer the following questions. Give an argument for your answer, either by providing a counter example or by giving a proof by induction. **Hint:** If you want to see more examples for proofs by induction then read on to Examples 6.11, 6.15 and 6.16. Compare this question with the one about concatenating strings, as part of Exercises 27–29.

- (a) Is the \( \oplus \) operation commutative?
- (b) Is the \( \oplus \) operation associative?
- (c) Does the \( \oplus \) operation have a unit? If yes, what is it? Justify your answer.

**Example 6.11.** The operations *sum* (see Examples 6.6 and 6.7) and \( \oplus \) (see Example 6.8) are defined above. We show by induction\(^3\) that for lists \( l \) and \( l' \) in \( \text{Lists}_N \) it is the case that

\[ \text{sum}(l + l') = \text{sum } l + \text{sum } l'. \]
### Base case list

\[
\begin{align*}
\text{sum}([] \mathbin{+} l') &= \text{sum} l' & \text{base case } + \\
&= 0 + \text{sum} l' & \text{0 unit for } + \\
&= \text{sum} [] + \text{sum} l' & \text{base case } \text{sum}.
\end{align*}
\]

### Step case list

\[
\begin{align*}
\text{sum}((s \mathbin{:} l) \mathbin{+} l') &= \text{sum}((s \mathbin{:} (l \mathbin{+} l'))) & \text{step case } + \\
&= s + \text{sum}(l \mathbin{+} l') & \text{step case } \text{sum} \\
&= s + (\text{sum} l + \text{sum} l') & \text{ind hyp} \\
&= (s + \text{sum} l) + \text{sum} l' & \text{associative} \\
&= \text{sum}(s \mathbin{:} l) + \text{sum} l' & \text{step case } \text{sum}.
\end{align*}
\]

### Tip

Defining a function recursively can be quite tricky when you are not used to it. To get started you might as well make use of the fact that you know what such a definition needs to look like. Assume we intend to define a function

\[f: \text{Lists}_N \rightarrow S\]

that behaves in a particular way. Then we know we need to have two cases:

- **Base case** \( f \). \( f[] =? \)
- **Step case** \( f \). \( f(s : l) =?? f.l. \)

It should be easy to read off from your description of \( f \) how to define it in the base case. For the step case you may require a bit of creativity. How does knowing \( f.l \) help us to calculate \( f(s : l) \)? Note that you are allowed to use operations from the target set of \( f \), here \( S \). If you can answer that question you should be able to give a correct definition. Then check your definition using an example.

Another useful operation is determining the length of a list. You are asked to define this operator yourself in the following exercise, but assume for the remainder of this section that there is a function

\[\text{len}: \text{Lists}_S \rightarrow \mathbb{N}\]

which, given a list, returns the number of elements in the list.

### Exercise 110.

This exercise concerns the length function described in the preceding paragraph.

(a) Give a definition of this \( \text{len} \) function.

---

\[^9\text{As previously discussed we do not state the induction hypothesis explicitly since its statement is identical to the statement we are proving.}\]
(b) Use your definition to calculate \( \text{len}[3, 2, 1] \) step by step.

(c) Give the code for the corresponding function for objects of type List.

(d) Show that for all lists \( l \) and \( l' \) over a set \( S \) we have
\[
\text{len}(l + l') = \text{len} l + \text{len} l'.
\]
Justify each step.

**Exercise 111.** For lists over \( \mathbb{N} \) give a recursive definition of a list being ordered,\(^{10}\) that is, for example for the list
\[
[k, l, m, n]
\]
we demand \( k \geq l \geq m \geq n \). *Hint: You want two base cases.*

We define additional operations for lists.

**Example 6.12.** We can also reverse a list, that is turn it back to front.

**Base case** rev.
\[
\text{rev} \ [\] = \ [\]
\]

**Step case** rev.
\[
\text{rev}(s : l) = \text{rev} l + + [s].
\]

You are asked to work out how this definition works in the following exercise.

**Exercise 112.** Carry out the following for the rev operator.

(a) Calculate \( \text{rev}[1, 0] \) step by step.

(b) Show that for every list \( l \) over an arbitrary set \( \text{len} \ l = \text{len} l \). *Hint: You may want to use a statement from a previous assessed exercise to help with this.*

(c) Show that \( \text{rev}(l + l') = \text{rev} l' + \text{rev} l \).

(d) Show that \( \text{rev rev} l = l \) for all lists \( l \) over \( S \). *Hint: You may want to show \( \text{rev}[s] = [s] \) separately.*

(e) Use the rev operator to give an alternative definition of the concatenation operator which uses recursion over the *second* argument. Give an argument that your definition agrees with the original.

Justify each step in your proofs.

**Example 6.13.** We give one more example where we look at code. When we have two lists it seems easy to decide whether they are ‘the same’. By that we mean they consist of the same elements in the same

\(^{10}\) We have chosen here to have the elements to get larger as they are added to the list—the opposite choice would also make sense.
order. A human being can check this for two elements of the set $\text{Lists}_S$ by inspection, and below we give a recursive procedure that carries out this check.

When we have to List objects, however, our idea of having ‘the same list’ may not agree with that of the Java programming language. The boolean value

$$(l_1 == l_2)$$

evaluates to true precisely when $l_1$ and $l_2$ refer to the same object. If we want to check whether two such objects consist of the same elements in the same order we have to write code of the following kind.

```java
public static boolean equal(List l1, List l2)
{
    if (l1 == null)
        return (l2 == null);
    else {
        if (l2 == null)
            return false;
        else
            return (l1.value == l2.value && equal(l1.next, l2.next));
    }
}
```

In our heads we are effectively defining a function that takes a List object and turns it into an element of $\text{Lists}_N$, and we consider two lists equal if this function maps them to the same list.

The code above can be turned into a mathematical definition quite easily. We want to define a function, say

$$f_=_: \text{Lists}_S \times \text{Lists}_S \rightarrow \{0, 1\} .$$

**Base case** $f_=_([], []).$ \hspace{1cm} $f_=([], []) = 1.$

**Base case** $f_=([], s': l').$ \hspace{1cm} $f_=([], s': l') = 0. $

**Base case** $f_=(s: l, []).$ \hspace{1cm} $f(s: l, []) = 0.$

**Step case** $f_=_.$

$$f_=(s: l, s': l') = \begin{cases} f_=(l, l') & s = s' \\ 0 & \text{else.} \end{cases}$$

where it is understood that $l$ and $l'$ are elements of $\text{Lists}_S$.

Why is the mathematical definition so much longer? Because in the code we make use of the equality function for integers and references.
Assume we have two functions

\[ f_{[]} : \text{Lists}_S \rightarrow \{0, 1\} \]

\[
\begin{array}{ll}
 l & \rightarrow \\
 1 & l = [] \\
 0 & \text{else}
\end{array}
\]

and

\[ f_S : S \times S \rightarrow \{0, 1\} \]

\[
\begin{array}{ll}
 (s, s') & \rightarrow \\
 1 & s = s' \\
 0 & \text{else.}
\end{array}
\]

then we can define instead

**Base case** \( f = f_{[]} \)

\[ f_{[]}([], l') = f_{[]} l'. \]

**Step case** \( f = [\]. \)

\[ f_{}(s : l, []) = 0. \]

**Step case** \( f = : s' : l'. \)

\[ f_{}(s : l, s' : l') = f_S (s, s') \land f_{}(l, l'), \]

which is very close to the code given above.

**Exercise 113.** The aim of this exercise is to define a function

\[ \text{search} : \text{Lists}_S \times S \rightarrow \{0, 1\} \]

\[
\begin{array}{ll}
 (l, s) & \rightarrow \\
 1 & s \text{ occurs in } l \\
 0 & \text{else.}
\end{array}
\]

(a) Recursively define this function.

(b) If your definition is a definition by cases then change that, using the \( f_S \) function from Example 6.13. **Hint:** You may use boolean operations on the set \( \{0, 1\} \).

(c) Write code that implements this function for objects of the class List.

**Example 6.14.** If we have a list \( l \) over a set \( S \), and a way of translating elements of \( S \) to elements of some set \( T \) via a function \( f : S \rightarrow T \) we may transform our list to one over the set \( T \) by using the function \( \text{map } f \) defined below. The type of this function is

\[ \text{map } f : \text{Lists}_S \rightarrow \text{Lists}_T. \]

It is given by the following recursive definition.

**Base case** \( \text{map } f \)

\[ (\text{map } f)[] = []. \]

**Step case** \( \text{map } f \)

\[ (\text{map } f)(s : l) = f s : (\text{map } f)l. \]
Assume we have the function \( f : \mathbb{N} \rightarrow \mathbb{Z} \) given by \( n \mapsto -n \).

Applying the \texttt{map} function to \( f \) and the list \([3, 2, 1]\) we get

\[
\begin{align*}
\text{(map } f \text{)}[3, 2, 1] &= \text{(map } f \text{)}(3 : [2, 1]) & \text{Step case list} \\
&= f3 : \text{(map } f \text{)}[2, 1] & \text{step case map} \\
&= -3 : \text{(map } f \text{)}(2 : [1]) & \text{def } f \text{ and step case list} \\
&= -3 : f2 : \text{(map } f \text{)}[1] & \text{step case map} \\
&= -3 : -2 : \text{(map } f \text{)}(1 : [1]) & \text{def } f \text{ and step case list} \\
&= -3 : -2 : (f1 : \text{map } f[1]) & \text{step case map} \\
&= -3 : -2 : (-1 : [1]) & \text{def } f, \text{ base case map} \\
&= -3 : -2 : [-1] \\
&= -3 : [-2, -1] \\
&= [-3, -2, -1].
\end{align*}
\]

We use the \texttt{map} operator to give additional examples for proofs by induction.

\textbf{Example 6.15.} We show that for the identity function \( \text{id}_S \) on a set \( S \) we have for all lists \( l \) over the set \( S \) that

\[
\text{(map } \text{id}_S \text{)}l = l.
\]

This is a proof by induction.

**Base case** list. We have \( \text{(map } \text{id}_S \text{)}[] = [] \) by base case map.

**Step case** list. We note the following.\(^{11}\)

\[
\begin{align*}
\text{(map } \text{id}_S \text{)}(s : l) &= \text{id}_S s : \text{(map } \text{id}_S \text{)}l & \text{step case map} \\
&= s : l & \text{id}_S s = s; \text{ ind hyp.}
\end{align*}
\]

\textbf{Example 6.16.} We show another property for the \texttt{map} operator. For two functions \( f : S \rightarrow T \) and \( g : T \rightarrow U \) we have for all lists \( l \) over \( S \) that

\[
\text{(map } (g \circ f) \text{)}l = \text{(map } g \text{)}(\text{map } f)l.
\]

We carry out the proof by induction.

**Base case** list.

\[
\begin{align*}
\text{(map } (g \circ f) \text{)}[] &= [] & \text{base case map} \\
&= \text{(map } g[1]) & \text{base case map} \\
&= \text{(map } g(\text{map } f[1])) & \text{base case map}.
\end{align*}
\]

\(^{11}\)As discussed above we do not spell out the induction hypothesis explicitly since its statement is identical to the statement we are proving.
Step case list.

\[(\text{map}(g \circ f))(s : l)\]
\[= (g \circ f)s : (\text{map}(g \circ f))l\]  
step case map
\[= g(fs) : (\text{map}g)((\text{map}f)l)\]  
\((g \circ f)s = g(fs), \text{ ind hyp}\)
\[= (\text{map}g)(fs : (\text{map}f)l)\]  
step case map
\[= (\text{map}g)((\text{map}f)(s : l))\]  
step case map.

Exercise 114. Use the map operator to solve the following problems. For each, define the function that you would like to apply the map operator to, and give an example of a list with at least three elements and show step-by-step how the required transformation is carried out.

(a) Turn a list of lengths in metres to one with the same lengths in kilometres.

(b) Turn a list containing numbers from 1 to 7 into a list of the corresponding days of the week.

(c) Turn a list of UK cities into a list of distances from the corresponding cities to Manchester (you don’t have to describe the function for this case formally—just describe the idea and give some of the values).

(d) Turn a list of course units in the School into a list of academic staff who are the leaders for the corresponding course units (you don’t have to describe the function for this case formally—just describe the idea and give some of the values).

Exercise 115. Show the following by induction, justifying each step.

(a) If \(l\) is a list over an arbitrary set \(S\) and \(f\) a function from \(S\) to some set \(T\) then \(\text{len}((\text{map}f)l) = \text{len}l\).

(b) If \(l\) and \(l'\) are elements of \(\text{Lists}_S\) and \(f : S \rightarrow T\) then
\[(\text{map}f)(l + l') = (\text{map}f)l + (\text{map}f)l'.\]

(c) If \(l\) is a list over an arbitrary set \(S\) and \(f\) a function from \(S\) to some set \(T\) then \((\text{map}f)(\text{rev}l) = \text{rev}(\text{map}f)l\).

(d) Let \(k_0\) be the function from \(\mathbb{N}\) to \(\mathbb{N}\) which maps every element to 0, that is
\[
k_0 : \mathbb{N} \rightarrow \mathbb{N}
\]
\[
n \mapsto 0.
\]
Assume that \(l\) is an element of \(\text{Lists}_\mathbb{N}\). Show by induction that for the list
\[(\text{map}k_0)l\]
every member is equal to 0.
Optional Exercise 25. This exercise explores further the connection between sets, lists over that set, and the map operator.

(a) Define a function $i_S$ from a set $S$ to lists over the set by mapping each element of $S$ to the one-element list containing just the element $s$, that is

$$i_S: s \mapsto [s].$$

Show that for all functions $f: S \to T$, and all $s \in S$ we have

$$(\text{map } f)(i_S s) = i_T(fs).$$

(b) Note that given a set $S$ there is nothing stopping us from building lists over the set of lists over $S$, that is, lists whose elements are lists (over $S$).

So given a function $f: S \to T$ we can use map map $f$ to map lists of lists over $S$ to lists of lists over $T$. Give some examples of how this operation works.

(c) Given a list of lists over $S$ we can ‘flatten’ this list into a list over $S$, for example by turning

$$[[2, 3, 4], [1, 5], [], [6]]$$

into

$$[2, 3, 4, 1, 5, 6].$$

Give a recursive definition of this ‘flattening’ operator, and let’s call this $j_S$.

(d) Show that for all lists $l$ over the set $S$ it is the case that

$$j_S((\text{map } i_S) l) = l.$$

(e) Show that for all functions $f: S \to T$, and all lists $L$ of lists over $S$ it is the case that

$$(\text{map } f)(j_S L) = j_T((\text{map map } f)L).$$

We have sketched here some advanced structure from an area called category theory which tells us something about how lists over a set relate to the original set.

6.2 Trees

A datatype that appears in many programming languages is that of a binary tree. You will see binary search trees as an example for a recursive algorithm in COMP162 (and those of you who have been to one of our Visit Days will have been introduced to the idea of binary search).
6.2.1 Binary trees defined

We give a formal definition for trees where every node has 0 or two children.

**Definition 47: tree**

A full binary tree with labels from a set $S$ is given by

**Base case** tree. For every element $s$ of $S$ there is a tree, $\text{tree } s$, consisting of just one node labelled with the element $s$.

**Step case** tree. Given two trees $t$ and $t'$ labelled over $S$, and an element $s$ of $S$, there is a tree $\text{tree}_s(t, t')$.

We use $\text{FBTrees}_S$ for the set of all binary trees with labels from $S$.

**Code Example 6.6.** Again we show how to define a corresponding class in Java, again for integers as the entries. To emphasize that this is a bit different from the mathematical notion defined above we use a slightly different name for this class.

```java
public class BTree {
    public int value;
    public BTree left, right;

    public BTree (int s, BTree t1, BTree t2)
    {value = s; left = t1; right = t2;}
}
```

Compare this to the following definition from *Java: Just in Time* (Section 22.11):

```java
public class BinTreeNode {
    private BinTreeNode left, right;
    private SomeClass nodeData;

    ...
}
```

Both these definitions of a class say that every binary tree node has two binary tree nodes, the left and the right one, and that it has a variable which contains data of some kind. In our case, the class `SomeClass` happens to be `int`, and with that instantiation we get our class `BTree`. 
How can we picture an object of this class? We use the same idea as the visualization of an object of class List. Assume we have

- a BTree object \( t \) with \( t\text{.value}=5 \) and \( t\text{.left}=t_2 \) and \( t\text{.right}=t_3 \) and
- a BTree object \( t_2 \) with \( t_2\text{.value}=3 \) and \( t_2\text{.left} = \text{null} \) and \( t_2\text{.right} = \text{null} \) and
- a BTree object \( t_3 \) with \( t_3\text{.value}=4 \) and \( t_3\text{.left}=t_4 \) and \( t_3\text{.right}=t_5 \) and
- a BTree object \( t_4 \) with \( t_4\text{.value}=2 \) and \( t_4\text{.left} = \text{null} \) and \( t_4\text{.right} = \text{null} \) and
- a BTree object \( t_4 \) with \( t_5\text{.value}=1 \) and \( t_5\text{.left} = \text{null} \) and \( t_5\text{.right} = \text{null} \) and

These objects may be pictured as follows, giving the whole tree in blue, the left subtree in red, and the right subtree (and its subtrees) in green.

See the following example for a less space-consuming way of drawing the corresponding full binary tree.

This kind of picture is a bit elaborate to draw, and in order to visualize instances of Definition 6.2 most people draw pictures such as the following for a tree with label \( s \) and left and right subtrees \( t \) and \( t' \) respectively.

\[
\begin{align*}
& s \\
\downarrow & \\
& t \quad t'
\end{align*}
\]

Note that \( t \) and \( t' \) are trees themselves, so the above does not give the full shape of the tree. To find that one has to look into how \( t \) and \( t' \) are defined.

**Example 6.17.** We look at the same tree as in Code Example 6.6, but this time we use the mathematical description.

---

12 There is one difference between the mathematical definition and the code below. Can you see what it is? Think about null references.
If we take the complete description of a tree, such as

\[ \text{tree}_5(\text{tree}_3, \text{tree}_4(\text{tree}_2, \text{tree}_1)) \]

over \( \mathbb{N} \), we may draw the full shape.

Note that in our definition there is no such thing as an empty tree. Every tree has at least one node.

We introduce some nomenclature for trees. For the tree \( \text{tree}_s(t, t') \) the node labelled with the \( s \) that appears in the description is the root of the resulting tree. In the above example that root has the label 5. The nodes labelled 3 and 4 are the children or the root node. Nodes that do not have any children are the leaves of the tree. Note that when you define a tree recursively then the leaves of the final tree are built by invoking the base case at the beginning of the building process. The inner nodes (that is, non-leaves) of the tree are built by invoking the step case.

In our definition the two maximal subtrees of a tree are given one after the other (in an ordered pair), and so it makes sense to speak of the first of these as the left (rooted at the node labelled 3) and the second as the right (rooted at the node labelled 4) subtree.

We illustrate these definition with the following illustrations.

---

For completeness’ sake we add two more definitions regarding trees. A binary tree is **perfect** if every leaf has the same distance to the root, namely the height of the tree. This is equivalent to demanding that on level \( i \), counting down, with the root giving level 0, there are \( 2^i \) many nodes.

A binary tree is **complete** if and only if there are \( 2^i \) nodes on level \( i \), with the possible exception of the final level, which must be ‘filled’ from left to right.

---

\(^{13}\)Note that elements of \( S \) may occur more than once in the tree—we give an example where this does not happen to make the connection with the picture clearer.
6.2.2 Operations on trees

Again we give examples for recursive definitions of operations for these recursively defined entities and give proofs by induction.

**Code Example 6.7.** For our Java class BTree we may want to know whether two objects describe the same tree structure. The following code does this for us.

```java
public static boolean equal (BTree t1, BTree t2)
{
    if (t1 == null)
        return (t2 == null);
    else {
        if (t2 == null)
            return false;
        else
            return (t1.value == t2.value)
                && equal (t1.left, t2.left)
                && equal (t1.right, t2.right);
    }
}
```

This method returns true precisely when the two trees providing its arguments have the same structure as trees.

The **height of a tree** gives a (crude) notion of the size of the tree.\(^{14}\) Compare the following definition with that of the length of a list, Exercise 110.

**Example 6.18.** We define the height function\(^ {15} \) recursively.

**Base case** \( hght \) tree \( s \) = 0.

**Step case** \( hght \) tree \( t, t' \) = \( \max \{ hght t, hght t' \} + 1 \).

We work out how this definition works for the sample tree from Example 6.17.

\[
\begin{align*}
\text{hght tree}_5(\text{tree}_3, \text{tree}_4(\text{tree}_2, \text{tree}_1)) &= \max\{\text{hght tree}_3, \text{hght tree}_4(\text{tree}_2, \text{tree}_1)\} + 1 & \text{step case hght} \\
&= \max\{0, \max\{\text{hght tree}_2, \text{hght tree}_1\} + 1\} + 1 & \text{def hght} \\
&= \max\{0, \max\{0, 0\} + 1\} + 1 & \text{base case hght} \\
&= \max\{0, 1\} + 1 & \text{def max} \\
&= 1 + 1 & \text{def max} \\
&= 2
\end{align*}
\]

\(^{14}\)Some people call this the **depth** of the tree.
Note that for this definition we have that the height of the tree is the integer part of the value of the logarithm (to base 2) of the number of nodes in the tree.

**Code Example 6.8.** We give the code that calculates the height of an object of class `BTree`.

```java
public static int height (BTree t)
{
    if (t.left == null && t.right == null)
        return 0;
    else
        return 1 + vMath.max(height(t.left), height(t.right));
}
```

What happens if this program is called for a tree that consists of a null reference? This is where the distinction between our mathematical binary trees and the `BTree` class becomes significant.

**Example 6.19.** We give a recursive definition of the function

\[ \text{no}: \text{FBTrees}_S \rightarrow \mathbb{N} \]

that counts the number of nodes in a tree in with labels from \( S \).

**Base case** \( \text{no} \). We set \( \text{no} (\text{tree}s) = 1 \).

**Step case** \( \text{no} \). We set \( \text{no} (\text{tree}_s(t, t')) = 1 + \text{no} t + \text{no} t' \).

Compare this definition with that of the length of a list, Exercise 110. We can show by induction that the number of nodes in a tree in the set \( \text{FBTrees}_N \) is odd.

**Base case** tree. We calculate \( \text{no} (\text{tree}s) = 1 \), which is odd.

**Ind hyp.** We assume that \( t \) and \( t' \) are binary trees each with an odd number of nodes.\(^{16}\)

**Step case** tree. We check this case.

\[ \text{no} (\text{tree}_s(t, t')) = 1 + \text{no} t + \text{no} t' \quad \text{step case} \ \text{no}. \]

By the induction hypothesis we can find \( k \) and \( k' \) in \( \mathbb{N} \) such that

\[ \text{no} t = 2k + 1 \quad \text{and} \quad \text{no} t' = 2k' + 1. \]

\(^{15}\)Note that when defining the height of a tree it makes sense to either count the number of layers in a tree, or to count the number of connections between them. If we want to former we set the height of a one-node tree as 1, and the latter case is given here.
Hence we may continue the above calculation as follows.

\[
\text{no}_{\text{tree}}(t, t') = 1 + \text{no } t + \text{no } t' \quad \text{step case } \text{no} \\
= 1 + 2k + 1 + 2k' + 1 \quad \text{ind hyp} \\
= 2(k + k' + 1) + 1 \quad \text{calcs in } \mathbb{N}
\]

This is an odd number which completes the proof.

Note that this is an example where proving the desired property requires us to do a bit more than write a sequence of equalities.

**Example 6.20.** We show by induction that for every tree \(t\) in the set \(\text{FBTrees}_N\) it is the case that \(\text{hght } t \leq \text{no } t\).

**Base case** tree.

\[
\text{hght}(\text{tree } s) = 0 \quad \text{base case } \text{hght} \\
\leq 1 \\
= \text{no}(\text{tree } s) \quad \text{base case } \text{no}.
\]

**Ind hyp.** For the trees \(t\) and \(t'\) we have

\[
\text{hght } t \leq \text{no } t \quad \text{and} \quad \text{hght } t' \leq \text{no } t'.
\]

We observe that for all natural numbers \(m\) and \(n\) we have that

\[
m \leq m + n \quad \text{and} \quad n \leq m + n,
\]

and so

\[
\text{max}\{m, n\} \leq m + n.
\]

For this reason the induction hypothesis implies

\[
\text{max}\{\text{hght } t, \text{hght } t'\} \leq \text{hght } t + \text{hght } t' \leq \text{no } t + \text{no } t',
\]

which we use below.

**Step case** tree.

\[
\text{hght}(\text{tree}_{s}(t, t')) \\
= \text{max}\{\text{hght } t, \text{hght } t'\} + 1 \quad \text{step case } \text{hght} \\
\leq \text{max}\{\text{no } t, \text{no } t'\} + 1 \quad \text{ind hyp} \\
\leq \text{no } t + \text{no } t' + 1 \quad \text{max}\{n, n'\} \leq n + n' \text{ in } \mathbb{N} \\
= \text{no } \text{tree}_{s}(t, t') \quad \text{step case } \text{no}.
\]

---

\[\text{This is the standard induction hypothesis for a data type with one step case which has two previously defined entities as inputs—we give it here because it's the first example of this kind.}\]
A proof by induction for trees in the set \( \text{FBTrees}_S \) always has the following shape. We are trying to prove a statement formulated in terms of the variable \( t \) which is an element of \( \text{FBTrees}_S \).

**Base case** tree. Prove the given statement for the case where all occurrences of \( t \) have been replaced by trees \( s \), where \( s \) is an arbitrary element of \( S \).

**Ind hyp** Assume the given statement holds for the trees \( t \) and \( t' \).

**Step case** tree. Prove the statement where all occurrences of \( t \) have been replaced by \( \text{tree}_s(t, t') \), where \( s \) is an arbitrary element of \( S \). The induction hypothesis is used as part of the proof.

Note how the general scheme derives from the shape of the definition of our trees.

### Code Example 6.9
We give one last version of code for a recursive function. Note that once again the difference between our mathematical binary trees, which cannot be empty, and the trees of class \( \text{BTree} \), which can, makes a difference.

```java
public static int no (BTree t)
{
if (t == null)
    return 0;
else
    return 1 + no(t.left) + no(t.right);
}
```

### Example 6.21
We give a recursive definition of the function

\[
\text{lvs}: \text{FBTrees}_S \rightarrow \mathbb{N}
\]

that counts the number of leaves in a tree in \( \text{FBTrees}_S \).

**Base case** \( \text{lvs} \).

\[ \text{lvs}\text{tree}_s = 1 . \]

**Step case** \( \text{lvs} \).

\[ \text{lvs}\text{tree}_s(t, t') = \text{lvs}t + \text{lvs}t' . \]

Note how little this definition differs from that of the function \( \text{no} \). A small change in a recursive definition can lead to a very different effect.

---

\(^{17}\)It may be necessary to assume it for \( t, t' \) and all their subtrees.
Tip

To get started on defining a recursive function for trees you might as well make use of the fact that you know what such a definition needs to look like. Assume we intend to define a function

$$f: \text{FBTrees}_N \rightarrow S$$

that behaves in a particular way. Then we know we need to have two cases:

**Base case** $f$. $f(\text{tree } n)$ =

**Step case** $f$. $f(\text{tree}_n(t, t')) = ??? f t \? f t'$.

It should be easy to read off from your description of $f$ how to define it in the base case. For the step case you may require a bit of creativity. How does knowing $f t$ and $f t'$ help us to calculate $f(\text{tree}_n(t, t'))$? Note that you are allowed to use operations from the target set of $f$, here $S$. When you have a definition check whether it does the right thing using an example.

**Exercise 116.** Assume that $N$ is a set of numbers, namely $\mathbb{Z}$, $\mathbb{Q}$ or $\mathbb{R}$.

(a) Draw the following tree: $\text{tree}_3(\text{tree}_2, \text{tree}_{-5}(\text{tree}_2, \text{tree}_{-1}))$

(b) Define a function

$$\text{sum}: \text{FBTrees}_N \rightarrow N$$

which sums up all the labels that occur in the tree (compare Example 6.6 where the analogous operation for lists is defined).

(c) Apply your function to the tree from part (a).

(d) Give the code for a function that performs the same job for an object of class BTree.

(e) Justifying each step show that if the set of labels is $N \setminus \{0\}$ then for all binary trees $t$ we have

$$\text{not } t \leq \text{sum } t.$$

**Exercise 117.** Consider the function

$$\text{search}: \text{FBTrees}_S \times S \rightarrow \{0, 1\}$$

which, for inputs $t$ and $s$, returns $1$ precisely when $s$ occurs as a label in the tree $t$.

(a) Give a recursive definition for this function, You may use the func-
tion \( f = g \) from Example 6.13 as well as the boolean operations on \( \{0, 1\} \).

(b) Give code that implements this function for objects of class \texttt{BTree}.

\textbf{CExercise 118.} Assume we have a function \( f : S \rightarrow T \).

(a) Recursively define a function \( \text{map}\ f \) that takes a binary tree with labels from \( S \) to a binary tree with labels from \( T \), that is

\[ \text{map}\ f : \text{FBTrees}_S \rightarrow \text{FBTrees}_T. \]

\textit{Hint: Consider the \text{map} function for lists as an example.}

(b) For the function

\[
\begin{align*}
    f : \mathbb{N} & \rightarrow \mathbb{N} \\
    n & \mapsto 2n
\end{align*}
\]

apply the function \( \text{map}\ f \) you defined in part (a) step by step to the following tree.

\[
\text{tree}_{17}(\text{tree}_{5}(\text{tree}_3, \text{tree}_19), \text{tree}_{25}(\text{tree}_19, \text{tree}_27))
\]

(c) Show that for every such function \( f \), and every tree \( t \) in the set \( \text{FBTrees}_S \), we have \( \text{hght}((\text{map}\ f)t) = \text{hght}\ t \).

(d) Show that for every such function \( f \) and every tree \( t \) in the set \( \text{FBTrees}_S \) we have \( \text{no}((\text{map}\ f)t) = \text{no}\ t \).

(e) Assume that \( \mathbb{N} \) is a set of numbers between \( \mathbb{N} \) and \( \mathbb{R} \). Show that for the function

\[
\begin{align*}
    k_1 : \mathbb{N} & \rightarrow \mathbb{N} \\
    n & \mapsto 1
\end{align*}
\]

we have for all binary trees \( t \) that \( \text{no}\ t = \text{sum}((\text{map}\ k_1)t) \).

Justify each step in your proofs.

\textbf{EExercise 119.} This exercise is concerned with defining a mathematical type that corresponds to the \texttt{BTree} class. We refer to such a tree as a \emph{binary tree with labels from a set} \( S \).

(a) Give a definition, similar to Definition 47, of a mathematical entity of \emph{binary trees} which corresponds to the \texttt{BTree} class.

(b) Recursively define an operation that takes two binary trees and returns a binary tree where the second argument has been added below the right-most leaf of the tree.

(c) Write code which implements the operation from the previous part.

\textbf{Exercise 120.} Give the following recursive definitions.

(a) Trees with labels from \( S \) such that every node has at most two
Children. Hint: You may want to allow an empty tree. You may also want to draw some examples to give you an idea what you are looking for.

(b) Trees with labels from $S$ such that every node can have an arbitrary finite number of children.

**Exercise 121.** This exercise is concerned with *perfect binary trees*.

(a) Recursively define the set of all those full binary trees with labels from a set $S$ that are perfect. Do so by describing, for each height, those elements of $\text{FBTrees}_S$ which are perfect. Hint: Have a look ahead to Example 6.42, and for each $n \in \mathbb{N}$ define a set $\text{PTrees}_S^n$ of those elements of $\text{FBTrees}_S$ which are perfect and have height $n$.

(b) Show that for such a tree $t$ we have

$$\text{lvs } t = 2^{\text{hght } t}.$$

(c) Show that it is indeed the case that, for $t \in \text{PTrees}_S^n$, we have $\text{hght } t = n$.

(d) Show that for a perfect tree $t$ we have $\text{lvs } t = 2^{\text{hght } t}$.

(e) Show that for a perfect tree $t$ we have

$$\text{no } t = 2^{1+\text{hght } t} - 1,$$

Conclude that

$$\text{hght } t = \log(1 + \text{no } t) - 1.$$

### 6.2.3 Ordered binary trees

Trees are useful structures when it comes to keeping data in a way that makes it easy to search. In Example 4.96 we looked at searching through an array, here we see in Exercise 123 how one can search through entries that are kept in a tree in an ordered way, rather than in an array.

**Example 6.22.** Sometimes we require the set of all the labels that occur in a tree, for example in order to define *ordered binary trees*, see below. The set of labels $\text{lab } t$ that occur in a tree $t$ with labels from $S$ is given as follows:

**Base case** $\text{lab }$. We have $\text{lab}(\text{tree } s) = \{s\}$.

**Step case** $\text{lab }$. We have $\text{lab}(\text{tree } s(t, t')) = \{s\} \cup \text{lab } t \cup \text{lab } t'$.

Note that this defines a function

$$\text{lab}: \text{FBTrees}_S \longrightarrow \mathcal{P}S,$$

---

18These binary trees are what the TreeSort class from *Java: Just in Time* is based on.
since it maps a binary tree with labels from \( S \) to a subset of \( S \).

If we have trees over a set of numbers \( N \) with \( \mathbb{N} \subseteq N \subseteq \mathbb{R} \) then that set comes with an order\(^{19}\) then we can recursively define what we mean by a binary tree over \( N \) being ordered.

**Definition 48: ordered binary tree**

A binary tree with labels from \( N \) is **ordered** under the following conditions.

**Base case** tree. A tree of the form \( \text{tree} \, n \) is ordered.

**Step case** tree. A tree of the form \( \text{tree} \, n \, (\text{tree} \, t, \text{tree} \, t') \) is ordered if and only if

- For every \( m \in \text{lab} \, t \) we have \( m \leq n \),
- for every \( m' \in \text{lab} \, t' \) we have \( n \leq m' \) and
- \( t \) and \( t' \) are ordered.

Ordered binary trees are sometimes called **binary search trees**.

We use \( \text{OFBTrees}_S \) for the set of ordered full binary trees with labels from the set \( S \).

Ordered binary trees do appear in *Java: Just in Time*, see Section 21.5. There the \( \text{TreeSet} \) class is used and you don’t have to program your own data structure for the resulting sorting operation. Can you tell a difference between our definition and the ordered binary tree used there? Ordered binary trees are used to keep an unknown number of data in a structure that allows for the binary search algorithm to be defined. Typically one would like to be able to insert new nodes, or to delete them. For this application one cannot use ordered full binary trees because after an insertion or deletion the tree will no longer be full, and one really should stick to ordered binary trees. Binary trees are mathematically defined in Exercise 119.

**Exercise 122.** Which of the following trees are ordered over the obvious set of numbers? **Hint:** You may find it easier to answer this question if you draw the tree first.

(a) \( \text{tree}_3(\text{tree} \, 2, \text{tree}_{17}(\text{tree} \, 5, \text{tree} \, 19)) \)

(b) \( \text{tree}_5(\text{tree}_2(\text{tree} \, 0, \text{tree} \, 1), \text{tree} \, 149)) \).

(c) \( \text{tree}_{17}(\text{tree}_5(\text{tree} \, 3, \text{tree} \, 19), \text{tree}_{25}(\text{tree} \, 19, \text{tree} \, 27)) \).

(d) \( \text{tree}_{25}(\text{tree}_9(\text{tree} \, 4, \text{tree} \, 1), \text{tree}_{49}(\text{tree} \, 36, \text{tree} \, 64)) \).

(e) \( \text{tree}_{-3}(\text{tree}_{-7}(\text{tree} \, -7, \text{tree} \, -4), \text{tree}_5(\text{tree} \, 4, \text{tree} \, 7)) \).

**Exercise 123.** Let \( N \) be a set of numbers between \( \mathbb{N} \) and \( \mathbb{R} \). Consider

\(^{19}\)See Section 7.4.1 for a formal definition of this concept.
the function

\[
\text{search} : \text{OFBTrees}_N \times N \rightarrow \{0, 1\}
\]

that for an input consisting of an ordered binary tree \( t \) with labels from \( N \) and an element \( n \) of \( N \) gives 1 if \( n \) occurs as a label in \( t \), and 0 otherwise.

(a) Give a recursive definition of \( \text{search} \). We may think of this function as performing a binary search.

(b) Write code that implements this function, assuming we have a class \( \text{OBTree} \) which restricts \( \text{BTree} \) to ordered binary trees only.

(c) Draw a picture of the ordered binary tree

\[
\text{tree}_3(\text{tree}_5(\text{tree}-7, \text{tree}-4), \text{tree}_5(\text{tree}_3, \text{tree}_7)).
\]

Imagine you want to add a node with label 4 to the tree. Where should that go? Draw your solution.\(^{20}\)

(d) Write code that takes as inputs an object of the class \( \text{OBTree} \) and an integer and returns an object of class \( \text{OBTree} \) in which the second argument has been inserted if it is not already present. Use the previous part to guide you.

*Hint: Your definition should take advantage of the fact that the tree is ordered and not look at every label.*

### 6.3 Syntax

Many formal languages are recursively defined. Examples of these are computer languages, or the language of logical propositions, or the language of regular expressions that appears in COMP11212.

This allows us to recursively define operations as well as to give inductive proofs of properties of such operations. This is one of the reasons why understanding recursion and induction is very useful in computer science. We describe some examples of this kind in this section.

#### 6.3.1 Strings

The simplest example of this kind is that of a string over a set \( S \). We also speak of a word over the alphabet \( S \).

**Base case** string. The empty string \( \epsilon \) is a string.

**Step case** string. If \( s \in S \) and \( w \) is a string then \( ws \) is a string.

One can now give a recursive definition of concatenation for strings, and one can define the length of a string recursively, just as these operations are defined for lists. Indeed, lists and strings are closely related.

\(^{20}\)Note that the result is not a full binary tree, but a binary tree, compare Exercise 119.

\(^{21}\)Since it’s a bit difficult to indicate something empty we use a symbol here. This is standard practice.
Exercise 124. Carry out the following tasks. This is a really nice exercise to work out whether you’ve understood all the concepts from this chapter.

(a) Give a recursive definition for the concatenation $+$ of strings over a set $S$. Ditto reusing $\text{len}$ and in particular $+$

(b) Give a recursive definition of the length $\text{len}$ of a string over a set $S$.

(c) Show that if $w$ and $w'$ are strings over a set $S$ then $\text{len}(w) + \text{len}(w') = \text{len}(w + w')$.

(d) Recursively define a function from lists over $S$ to strings over $S$.

(e) Recursively define a function from strings over $S$ to lists over $S$.

(f) Show by induction that your two functions are mutual inverses of each other.

(g) What can you say about your two functions regarding how they behave with respect to the $\text{len}$ functions for strings and lists? What about these functions and concatenating strings, versus concatenating lists?

Justify each step in your proofs.

6.3.2 Logical propositions

We now turn to the language of logical propositions that is studied in Chapter 3. We recall the definition of the boolean interpretation of a propositional formula from Section 3.2.1.

Base case prop. For every propositional variable $Z$, we have that $Z$ is a proposition.

Step cases prop. Assume that $A$ and $B$ are propositions. Then the following are propositions.

\begin{align*}
\text{Step case } \neg &. \quad (\neg A), \\
\text{Step case } \land &. \quad (A \land B), \\
\text{Step case } \lor &. \quad (A \lor B) \text{ and} \\
\text{Step case } \to &. \quad (A \to B).
\end{align*}

Example 6.23. The definition of an interpretation relative to a valuation of a proposition is also via recursion. Assume that we have a valuation $v$ that gives a value to each propositional variable in the boolean algebra $\{0, 1\}$. We give the formal definition of the boolean interpretation of propositions. This is given by a function $I_v$ which maps a proposition to its boolean interpretation relative to $v$. The source of this function is the set of all propositions, and its target is $\{0, 1\}$.

\footnotesize
\begin{itemize}
\item[22] Below we use the shorter ‘proposition’ instead of ‘propositional formula’.
\item[23] In this section we use this letter for propositional variables.
\end{itemize}
Base cases $I_v$. For a propositional variable $Z$ we define $I_v Z = v Z$.

Step cases $I_v$. We define:

- **Step case $\neg$**: $I_v (\neg A) = \neg I_v A$,
- **Step case $\land$**: $I_v (A \land B) = I_v A \land I_v B$,
- **Step case $\lor$**: $I_v (A \lor B) = I_v A \lor I_v B$ and
- **Step case $\rightarrow$**: $I_v (A \rightarrow B) = I_v A \rightarrow I_v B$.

Note that to argue about the properties of semantic equivalence studied in the material on logic we did not use induction. To show, for example, that the boolean interpretation of $\neg \neg A$ is the same as that of $A$ for all valuations is, when fully spelled out, an induction proof but it can be made plausible without that. See the following exercise for an idea of how this works formally.

**Tip**

A proof by induction for propositions usually takes the following shape:

**Base cases** prop. We show that the statement holds for every proposition of the form $Z$, where $Z$ is a propositional variable.

**Ind hyp.** We assume that the statement holds for formulae $A$ an $B$.

**Step cases** prop. We show that, given the induction hypothesis, the statement holds for formulae of the form

- $\neg A$,
- $A \land B$,
- $A \lor B$,
- $A \rightarrow B$.

**Exercise 125.** Give the formal definition of the powerset interpretation for the set of all propositions (see Section 3.2.2).

**Exercise 126.** Solve the following problems for propositional logic.

(a) Give a recursive definition of the function $\text{var}$ which takes as its argument a propositional formula and returns the set of propositional variables that occur in that formula.

(b) Give a recursive definition of the subformula construction as a function $\text{subf}$ which takes as its argument a propositional formula and returns the set of all its subformulae, see Section 3.1.

(c) Justifying each step, show by induction that for all propositional formulae $A$ we have

\[ \text{var } A \subseteq \text{subf } A. \]
Note that the definition of a formal derivation in a natural deduction system is also recursive: The base cases are given by the axiom rules and the step cases by the other derivation rules.

6.3.3 Formal languages

We look at elements of formal languages which appear on COMP11212.

Regular expressions

The following explains a recursive definition from Part 1 of COMP11212.

**Example 6.24.** Let $\Sigma$ be a set of symbols. A **pattern** or **regular expression over** $\Sigma$ is generated by the following recursive definition.

- **Base case** $\text{regexp } \emptyset$. The character $\emptyset$ is a pattern;
- **Base case** $\text{regexp } \epsilon$. the character $\epsilon$ is a pattern;
- **Base case** $\text{regexp } \Sigma$. every symbol from $\Sigma$ is a pattern;
- **Step case** $\text{regexp } +$. if $p_1$ and $p_2$ are patterns then so is $\text{regexp } (p_1p_2)$;
- **Step case** $\text{regexp } |$. if $p_1$ and $p_2$ are patterns then so is $\text{regexp } (p_1|p_2)$;
- **Step case** $\text{regexp } *$. if $p$ is a pattern then so is $\text{regexp } (p*)$.

It is typical for formal languages that there are several base cases as well as several step cases. This is due to the fact that we have several notions of a ‘primitive’ term in our language (that is one that is not built from other terms), and we have several ways of putting terms together to get new terms.

**Example 6.25.** There is a related recursive definition which tells us when a string made up of symbols from $\Sigma$ **matches** a given regular expression.

Let $p$ be a pattern over a set of symbols $\Sigma$ and let $s$ be a string consisting of symbols from $\Sigma$. We say that $s$ **matches** $p$ if one of the following cases holds:

- **Base case** $\text{regexp } \epsilon$. The empty word $\epsilon$ matches the pattern $\epsilon$.
- **Base case** $\text{regexp } \Sigma$. A character $x$ from $\Sigma$ matches the pattern $p = x$.
- **Step case** $\text{regexp } +$. The pattern $p$ is a concatenation $p = (p_1p_2)$ and there are words $s_1$ and $s_2$ such that $s_1$ matches $p_1$, $s_2$ matches $p_2$ and $s$ is the concatenation of $s_1$ and $s_2$.

---

24You will note that some people use formulas, and some people formulae, for the plural. The latter is the original Latin form, but in modern English some prefer to form an English plural.

25The name of this step case mentions an operator that does not appear in the term constructed, but it’s not clear what better name there is for concatenation.
Step case regexp \(|\). The pattern \(p\) is an alternative \(p = (p_1|p_2)\) and \(s\) matches \(p_1\) or \(p_2\) (it is allowed to match both).

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

Noticeably there is no entry for Base case regexp \(\emptyset\), and the reason for this is that no string matches the pattern \(\emptyset\). Consequently there is no need to include that base case.

Example 6.26. A result that is given in the notes for COMP112, but not proved, is covered in the following (somewhat extended) example. Given a regular expression \(p\), we recursively define

\[
L(p)
\]
as follows:

- **Base case** regexp \(\emptyset\). \(L(\emptyset) = \emptyset\).
- **Base case** regexp \(\epsilon\). \(L(\epsilon) = \{\epsilon\}\).
- **Base case** regexp \(\Sigma\). \(L(x) = \{x\}\) for all \(x \in \Sigma\).
- **Step case** regexp \(+\). \(L(p_1p_2) = L(p_1) \cdot L(p_2)\).
- **Step case** regexp \(\mid\). \(L(p_1|p_2) = L(p_1) \cup L(p_2)\).
- **Step case** regexp \(*\). \(L(p^*) = (L(p))^*\).

The operations \(\cdot\) and \((-)^*\) for sets of strings are defined in the COMP112 notes.

Note that we can think of \(L\) as a function from the set of all regular expressions over the alphabet \(\Sigma\) to the powerset of the set of strings over \(\Sigma\), \(P\Sigma^*\).

The claim made, but not proved, in the COMP112 notes is

**Proposition**

For all regular expressions \(p\) over the alphabet \(\Sigma\) we have that

\[
L(p) = \{s \in \Sigma^* \mid s \text{ matches } p\}.
\]

**Proof.** One may prove this by induction. Note that the induction proof has the number of base and step cases indicated by the recursive definition of a regular expression.

- **Base case** regexp \(\emptyset\). \(L(\emptyset) = \emptyset\). Since no string matches the pattern \(\emptyset\) the set of strings that match this pattern is indeed the empty set.
**Base case** regexp $\epsilon$. $L(\epsilon) = \{\epsilon\}$. By definition, $\epsilon$ is the only string that matches the pattern $\emptyset$.

**Base case** regexp $\Sigma$. $L(x) = \{x\}$ for all $x \in \Sigma$. By definition the string $x$ is the only string that matches the pattern $x$.

**Step case** reg-ex $\cdot$. $L(p_1p_2) = L(p_1) \cdot L(p_2)$. By definition of $\cdot$ for sets of words we have that,

$$L(p_1) \cdot L(p_2) = \{s_1s_2 \mid s_1 \in L(p_1), s_2 \in L(p_2),$$

By the induction hypothesis, this latter set is the set of all concatenations of strings $s_1s_2$ such that

$s_1$ matches $p_1$ and $s_2$ matches $p_2$,

and by definition, a string $s$ matches the pattern $p_1p_2$ if and only if there exist strings $s_1$ and $s_2$ such that

$s_1$ matches $p_1$ and $s_2$ matches $s$ and $s = s_1s_2$,

which gives the claim.

**Step case** reg-exp $|$. $L(p_1|p_2) = L(p_1) \cup L(p_2)$. By definition a string $s$ matches $p_1|p_2$ if and only if at least one of

$s$ matches $p_1$ or $s$ matches $p_2$,

which by the induction hypothesis means that this is the case if and only if

$s \in L(p_1) \cup L(p_2)$.

**Step case** reg-exp $\ast$. $L(p^\ast) = (L(p))^\ast$. By definition the pattern $p^\ast$ is matched by the string $s$ matches if and only if there is

$n \in \mathbb{N}$ and $s_1, s_2, \ldots s_n \in \Sigma^\ast$

with

$s = s_1s_2 \cdots s_n$

and $s_i$ matches $p$ $(0 \leq i \leq n)$.

But by the induction hypothesis, this means that

$s_i \in L(p)$ for $0 \leq i \leq n$,

and this is equivalent to

$s = s_1s_2 \cdots s_n \in (L(p))^\ast$

as required.
In the notes for COMP11212 no formal proof is given that for every regular expression there is a finite state automaton defining the same language—what is described there is effectively a sketch of such an argument. Such a proof would proceed by induction over the above definition, including all the given cases. If you look at the text that explains how to turn a regular expression into a finite state automaton you should be able to see how each base case, and each step case, is treated there. As a consequence the essence of the proof is given in those notes, and all you would have to do to complete it is to introduce the formal induction structure.

**Context-free grammars**

There are a number of recursive definitions connected with context-free grammars. If we look at the definition of a string being generated by a grammar we can see that these are instructions that follow recursive rules, using the following pattern:

**Base case** $\Gamma$. This is provided by the start symbol $S$.

**Step cases** $\Gamma$. We have a step case for each production rule.

If we generalize the definition of a string being generated by a grammar to strings consisting of both, terminal and non-terminal, symbols the recursive nature of the construction becomes obvious.

**Example 6.27.** The definition of when a string is generated by a grammar is one example:

**Base case** gen. by $\Gamma$. The string $S$ is generated by $\Gamma$.

**Step cases** gen. by $\Gamma$. If $R \rightarrow Y$ is a rule of the grammar and the string $X$ is generated by $\Gamma$ then the string that results from replacing one occurrence of $R$ in $X$ by $Y$ is a string generated by $\Gamma$.

**Example 6.28.** Similarly the notion of a parse tree for a grammar has an underlying recursive definition:

**Base case** parse tree for $\Gamma$. The tree consisting of only one node, which is labelled $S$, is a parse tree for $\Gamma$.

**Step cases** parse tree for $\Gamma$. If $R \rightarrow Y$ is a rule of the grammar and the tree $t$ is generated by $\Gamma$ then the tree that results from the following process is a parse tree for $\Gamma$:

- Identify a leaf labelled $R$.
- Add new children to that leaf, one for every symbol that occurs in $Y$ in that order.

For example assume we have a grammar with non-terminal symbols $S$,
$U$ and $V$, and terminal symbols digits from 0 to 9, if we have a parse tree as follows.

![Parse tree diagram]

and a rule

$$U \rightarrow 1V2$$

then both of the following trees are parse trees for $\Gamma$:

![Parse tree diagram]

### 6.3.4 Elements of programming languages

**Example 6.29.** To give another example, assume that we have a set of numbers $N$, on which we have various operations and properties, see below for a concrete example. For such a set of numbers we may express calculations by forming terms that consist of various elements of $N$ connected by the available operations. This gives us a language of such expressions. Many programming languages have such a notion of *arithmetic expressions* built in.

One might try to specify an arithmetic expression over the set $N$ by the following recursive definition:

**Base cases** are $\text{ex}$. For every $n \in N$ we have that $n$ is an arithmetic expression.

**Step cases** are $\text{ex}$. Assume that $a$ and $a'$ are arithmetic expressions. Then the following are arithmetic expressions:

- **Step case** $()$. $(a)$,
- **Step case** $\text{--}$. $-a$,

---

$^{26}$In the COMP11212 notes the only parse trees drawn are those whose leaves are labelled with terminal symbols, but for a formal definition one needs to first allow a more general version.
<table>
<thead>
<tr>
<th>Step case</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>−1</td>
<td>$a^{-1}$</td>
</tr>
<tr>
<td>+</td>
<td>$a + a'$</td>
</tr>
<tr>
<td>·</td>
<td>$a \cdot a'$</td>
</tr>
</tbody>
</table>

This is the kind of definition (see the While language below) that appears as part of a larger definition of a programming language. If the set of numbers in question is some approximation to the real numbers then one might also have built-in operations such as the trigonometric functions, and there is probably support for exponentiation of numbers.

Typically the idea is that the programmer, as part of the code, uses arithmetic expressions to specify calculations to be carried out by the computer at run-time.

There’s one problem with the definition from Example 6.29: Not all arithmetic expressions defined in this way work as intended: This is because we have done nothing to ensure that the multiplicative inverse is only applied to expressions which are different from 0. This means that at run-time (or in some cases at compile-time) the machine will find itself being asked to carry out a multiplication by an inverse that does not exist, such as a division by 0 in the integers.

One would like to change the above definition by specifying a changed step case:

- If $a$ is an arithmetic expression with $a \neq 0$ then $a^{-1}$ is an arithmetic expression.

But in order to carry out the test to check whether an arithmetic expression is equal to 0, either the programmer, or the machine, would effectively have to do all the required calculations before the program is compiled. Clearly it would be pointless to expect the programmer to carry out this check when writing the program. After all, the point of writing code is to have the computer deal with such calculations. But it turns out that one can not ask the computer to do these calculations prior to run-time either. This is an issue related to the Halting Problem, which is explained in more detail in COMP11212.

Every programming language that supports arithmetic expressions has a recursively defined evaluation procedure for these expressions, which tell the machine how to calculate the value of such an arithmetic expression (or throw up an error if a division by 0 occurs).

For most programming languages the valid programs are defined recursively, typically using grammars. For most languages this definition is very large. Typically it involves defining more than one kind of entity. Below we give an example of a language that has a short definition to given an idea of how this might work. Note that this language is no less powerful (in a formally definable way) than, say, Java.

### Example 6.30

We give the definition of the While programming language as it appears in the notes for COMP11212. First it is necessary to define arithmetic expressions for While. This

\[^{27}\text{Typically there are more expressions, for example the programmer is allowed to write } a - a' \text{ as a shortcut for } a + (-a').\]
assumes that there is a notion of number predefined, and that we have a notion of variable. To keep this definition short we do not cover those here.  

**Base case** $AExp\ n$. For every number $n$ we have that $n$ is a While arithmetic expression.

**Base case** $AExp\ var$. For every variable $x$ we have that $x$ is a While arithmetic expression.

**Step case** $AExp\ +$. If $a_1$ and $a_2$ are While arithmetic expressions then so is $a_1 + a_2$.

**Step case** $AExp\ −$. If $a_1$ and $a_2$ are While arithmetic expressions then so is $a_1 - a_2$.

**Step case** $AExp\ ×$. If $a_1$ and $a_2$ are While arithmetic expressions then so is $a_1 \times a_2$.

These arithmetic expressions are used in the following definition of While boolean expression:

**Base case** $BExp\ True$. There is a While boolean expression $True$.

**Base case** $BExp\ False$. There is a While boolean expression $False$.

**Step case** $BExp\ =$. If $a_1$ and $a_2$ are While arithmetic expressions then $a_1 = a_2$ is a While boolean expression.

**Step case** $BExp\ ≤$. If $a_1$ and $a_2$ are While arithmetic expressions then $a_1 \leq a_2$ is a While boolean expression.

**Step case** $BExp\ ¬$. If $b$ is While boolean expressions then so is $¬b$.

**Step case** $BExp\ ∧$. If $b_1$ and $b_2$ are While boolean expressions then so is $b_1 \land b_2$.

Finally we can define While statements.

**Base case** $Stm\ :=$. If $x$ is a variable and $a$ is a While arithmetic expression then $x := a$ is a statement.

**Base case** $Stm\ skip$. There is a statement $skip$.

**Step case** $Stm\ ;$. If $S_1$ and $S_2$ are statements then so is $S_1; S_2$.

**Step case** $Stm\ if$. If $b$ is a While boolean expression and $S_1$ and $S_2$ are statements then $if\ b\ then\ S_1\ else\ S_2$ is a statement.

**Step case** $Stm\ while$. If $b$ is a While boolean expression and $S$ is a statement then $while\ b\ do\ S$ is a statement.
This is not how a computer scientist would write down the definition. You can see that it is very verbose, and that many of the words are not really required to understand what is being defined. For this reason computer scientists have created their own notations for making such definitions more compact, and you will see some of those in COMP11212. But it is only the notation that differs—mathematically speaking, the corresponding definitions are still examples of recursive definitions. When we have a formal definition of a programming language we can then reason about all programs for that language. One area of computer science where this happens is the semantics of programming languages.

Also note that the definition from Example 6.30 only tells us what the valid While programs are—it does not tell us anything about how those should be executed, and what the result of any computation should be. You will see how one can do that, again making use of a recursive definition, in the second part of COMP11212.

6.4 The natural numbers

The classic example of a recursively defined set is that of the natural numbers, and it is the only way of formally defining what this set should be. Very little has to be assumed for this definition.

6.4.1 A formal definition

\textbf{Definition 49: natural numbers}

The elements of the set of natural numbers \( \mathbb{N} \) are given by the following recursive definition.

\textbf{Base case} \( \mathbb{N} \). There is a natural number 0.

\textbf{Step case} \( \mathbb{N} \). For each natural number \( n \) there is a natural number \( Sn \), the successor of \( n \).

It is more customary to write \( n + 1 \) for \( Sn \), but strictly speaking this has to be justified (see Exercise 128 (b)), and we use \( Sn \) for the time being. We introduce

\[ 1 \]

as a shortcut for \( So \).

\textbf{Code Example 6.10.} We give a Java class that corresponds to this definition.

\begin{verbatim}
public class Nat {
    public Nat next;
}
\end{verbatim}
This is a very odd class that one wouldn’t want to use in practice, but the example here is to show you that this can be done. Note that this class does not have a built-in upper bound for the numbers so defined (unlike the Java class List). In practice the size of the numbers you can use in this way is limited by the machine’s memory. Note also the way the class is constructed the number you get from using the argument encoding \( n \) is the number encoding \( n + 1 \).

You can picture an object of this class as shown below. Assume that we have

- a Nat object \( n \) with \( n.\text{next}=\text{n3} \) and
- a Nat object \( \text{n3} \) with \( \text{n3.}\text{next}=\text{n2} \) and
- a Nat object \( \text{n2} \) with \( \text{n3.}\text{next}=\text{n1} \) and
- a Nat object \( \text{n1} \) with \( \text{n1.}\text{next}={}\text{null} \).

You may picture these objects as follows:

![Diagram of Nat objects]

Just as we need to count the number of symbols \( S \) to know which number is given by

\[ SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS0, \]

so we have to check how many references we have to follow until we reach the \text{null} reference to know which number is given by some object of class Nat.

Previously most of you will have assumed that the natural numbers exist, and that you know what they are, and how their operations work, without having seen a formal definition. For now you are expected to work with this formal definition.

### 6.4.2 Formal definitions for basic operations on the natural numbers

In what follows we illustrate how the familiar operations may be formally defined. You could also think of the following as something that could be implemented on a fairly primitive computer.
Example 6.31. An example of a recursively defined operation is addition. The sum $m + n$ of two natural numbers $n$ and $m$ is defined as follows. For all $m$ in $\mathbb{N}$:

**Base case** +. \[ m + 0 = m \]

**Step case** +. \[ m + Sn = S(m + n) \]

We have defined $m + n$ by recursion over the argument $n$, covering the base case and the step case for natural numbers. Note that in the step case we are using the fact that $m + n$ may be assumed to be already defined when giving a definition for $m + Sn$. Further note that nothing is assumed about the nature of $+$, but, of course, our definition here gives the common notion of addition.

We could just as well have used as our definition. Since we are aiming to define the usual addition operator, we expect the result to be commutative, and so it should not matter which argument we recurse over. We show below that this definition does indeed give a commutative operator.

Note that the official definition of addition is the first one given—this is the one you must use in the exercises.

Code Example 6.11. We give the code for this operation for the class defined above.

```java
public static Nat plus (Nat n, Nat n2)
{
    if (n == null)
        return n2;
    else
        return new Nat (plus(n.next,n2));
}
```

Now assume that we had a class `nat` in Java like `int`, but restricted to the natural numbers. Then you could write the following code to implement addition.

```java
public static nat plus (nat n, nat n2)
{
```

---

30 Compare this with the definition of concatenating two lists, Example 6.8.
31 This is where the situation here differs from the $+$ operation for lists.
32 You may want to compare this definition with that of concatenating two lists, Example 6.8.
if (n == 0) {
    return n2;
} else {
    return 1 + (plus(n−1,n2));
}

Obviously one wouldn’t write this in practice, but does work and implements addition.

**Exercise 127.** Assume you would prefer your natural numbers to start with 1 rather than 0.

(a) Give a definition for the natural numbers starting with 1.

(b) Give code for a corresponding class.

(c) Give a definition for addition.

(d) Give the code for an addition operation for your class from part (b).

**Example 6.32.** Once again we may use induction to prove properties of the operation just defined. For a very simple example let us prove that for all natural numbers \( n \) we have
\[
0 + n = n.
\]
Here the base case is for \( n = 0 \), and the step case is for \( Sn \), and we may use the induction hypothesis \( 0 + n = n \) for the latter. This is what the formal proof looks like.

**Base case** \( \mathbb{N} \). We note \( 0 + 0 = 0 \) by base case +.

**Ind hyp.** We assume that we have \( 0 + n = n \).

**Step case** \( \mathbb{N} \). We calculate
\[
0 + Sn = S(0 + n) \quad \text{step case +}
\]
\[
= Sn \quad \text{induction hypothesis}.
\]

**Tip**

The standard structure for an inductive proof over the natural numbers is the following. Assume we have a statement given in terms of the variable \( n \) denoting an element of \( \mathbb{N} \).

**Base case.** We show the statement for some number \( b \). The base case is the case of the smallest number \( b \) for which the given statement holds, and it is obtained by replacing every occurrence of \( n \) by that smallest number.

**Ind hyp** We assume that the statement holds for \( n \) (in which case it

---

33 Compare this with the code for concatenating two lists, Code Example 6.5.
coincides with the given statement), or possibly all numbers that are less than or equal to \( n \), and greater than or equal to the number \( b \) from the base case.

**Step case.** We show the statement for \( Sn \) by proving the statement where every occurrence of \( n \) has been replaced by \( Sn \).

Below we do not give the induction hypothesis explicitly if it coincides with the statement we are proving.

**Example 6.33.** As another example we show that

\[
m + Sn = Sm + n. \quad (*)
\]

Often in a statement with two variables where induction is required it is sufficient to carry out the induction over one of the variables. Usually the better choice is to follow the pattern given by the recursive definition, here the definition of \( + \), and so we carry out an induction over the variable \( n \).

Note that for more complicated statements it may be necessary to carry out a double induction proof, that is, a proof where induction has to be carried out over both arguments.

We begin the proof of statement \((*)\) with the base case, \( n = 0 \).

**Base case** \( \mathbb{N} \). We have

\[
\begin{align*}
m + S0 &= S(m + 0) \quad \text{step case +} \\
       &= Sm \quad \text{base case +} \\
       &= Sm + 0 \quad \text{base case +}.
\end{align*}
\]

Next one has to establish the step case, where \( Sn \) appears in the place of \( n \), and where one may use the induction hypothesis, that is the claim \( Sm + n = m + Sn \).

**Step case** \( \mathbb{N} \). We have

\[
\begin{align*}
m + SSn &= S(m + Sn) \quad \text{step case +} \\
       &= S(Sm + n) \quad \text{induction hypothesis} \\
       &= Sm + Sn \quad \text{step case +}.
\end{align*}
\]

What is the point of property \((*)\)? It turns out to be very useful, see below, and it will be helpful for you in Exercise 128. It takes a statement where we cannot apply the step case of the definition of \( + \) and turns it into a statement where we can. This is how it is used below.

**Example 6.34.** As a more complicated example we show that addition is commutative, that is

for all \( m, n \in \mathbb{N} \) we have \( m + n = n + m \).
**Base case** $\mathbb{N}$. This is established as part of Exercise 128 (a).

**Step case** $\mathbb{N}$. We have

$$m + Sn = S(m + n) \quad \text{step case} +$$
$$= S(n + m) \quad \text{induction hypothesis}$$
$$= n + Sm \quad \text{step case} +$$
$$= Sn + m \quad \text{property (∗)}.$$

**CExercise 128.** This exercise is concerned with the properties of addition. For this exercise use the original definition of addition by recursion over the second argument—do not switch to the alternative mentioned in Example 6.31. Justify each step.

(a) Show that 0 is a unit for addition on the natural numbers.

(b) Show that $m + 1 = Sm = 1 + m$ for all $m \in \mathbb{N}$, where 1 is a shortcut for $S0$.

(c) Show that addition on the natural numbers is associative.

Defining further operations for the natural numbers, for example multiplication, is now possible. The more complicated the operation, the more complicated is the formal definition likely to be, and the same goes for proving properties of such operations. However, it is possible to use operations previously defined, and so one can build up complicated operations step by step.

**Example 6.35.** The product $m \cdot n$ of two natural numbers $m$ and $n$ is defined as follows. For all $m \in \mathbb{N}$ we have

**Base case** $\cdot$. \quad $m \cdot 0 = 0$

**Step case** $\cdot$. \quad $m \cdot Sn = m \cdot n + m$.

We recurse over the first variable here, just to show that it does not really matter for an operation that can be shown to be commutative.

**Exercise 129.** This exercise is concerned with properties of multiplication. Justify each step.

(a) Show that 1 is a unit for multiplication on the natural numbers.

(b) Show that for all $m \in \mathbb{N}$ we have

$$m \cdot 0 = 0 = 0 \cdot m.$$

(c) Show that for natural numbers $m$ and $n$ we have $Sm \cdot n = m \cdot n + n$.

(d) Show that multiplication for natural numbers is a commutative operation.
(e) Show that we have the following distributivity law for natural numbers:
\[ k \cdot (m + n) = k \cdot m + k \cdot n. \]

(f) Show that multiplication for natural numbers is associative.

**CExercise 130.** Carry out the following tasks, justifying each step in your proofs.

(a) Define a function \( d \) from \( \mathbb{N} \) to \( \mathbb{N} \) that doubles its argument, without referring to the operations \(+\) or \(\cdot\).

(b) Give code that implements this operation for the class `nat` from Code Example 6.10.

(c) Show that for all \( n \in \mathbb{N} \) we have \( d_n = S0 \cdot n \). *Hint: You may find property (\(\ast\)) useful, and you will definitely require addition in this proof.*

(d) Consider the function \( \text{mod}_2 \) defined as follows:

- **Base case** \( \text{mod}_2 : 0 \rightarrow 0 \).
- **Base case** \( \text{mod}_2 : S0 \rightarrow S0 \).
- **Step case** \( \text{mod}_2 : \text{mod}_2(SSn) = \text{mod}_2(n) \).

How would you describe the action of this function? Why do we need two base cases to define this function? Show that for all \( n \in \mathbb{N} \) we have \( \text{mod}_2 d_n = 0 \).

**Exercise 131.** For natural numbers \( m \) and \( n \) define \( m^n \). You are allowed to use operations defined above in this chapter. Then prove that for all \( n, k \) and \( l \) in \( \mathbb{N} \) we have \( n^{k+l} = n^k \cdot n^l \), justifying each step. You may use results from previous exercises as needed.

**Example 6.36.** We can define the predecessor \( P_n \) of a natural number \( n \) as follows.

- **Base case** \( P \). \( P0 = 0 \).
- **Step case** \( P \). \( P(Sn) = n \).

**Code Example 6.12.** Again we give code that implements this operation for the class `nat`.

```java
public static Nat pred (Nat n)
{
    if (n == null)
        return n;
```
Example 6.37. The notion of a predecessor allows us to define an operation related to subtraction for the natural numbers, which satisfies
\[ m \dot{-} n = 0 \quad \text{if} \quad m < n, \]
that is we use the value 0 wherever subtraction in the integers would lead to a negative number. In other words we give a recursive definition of the function
\[ m \dot{-} n = \begin{cases} m - n & \text{if} \quad m \geq n \\ 0 & \text{else} \end{cases} \]

**Base case** \( \dot{-} \) \( m \dot{-} 0 = m \)

**Step case** \( \dot{-} \) \( m \dot{-} Sn = P(m \dot{-} n) \).

Note that there is no sensible way of defining this operation by recursing over the first argument: The reason our definition works is that the result of \( m \dot{-} Sn \) is related to that of \( m \dot{-} n \): The result of \( m \dot{-} Sn \) should be one below that of \( m \dot{-} n \), unless the result of \( m \dot{-} n \) is 0 already, in which case \( m \dot{-} Sn \) must be 0 as well. This is exactly how the predecessor function \( P \) works. There is no such simple relation between \( Sm \dot{-} n \) and \( m \dot{-} n \).

Optional Exercise 26. Think about what would be required to define the \( \dot{-} \) operator by recursing over the first argument. You may find that you need to define \( \leq \) for natural numbers, see Exercise 132.

Example 6.38. Recall integer division from page 16. This is more complicated to define formally since we require case distinctions. For natural numbers \( m \) and \( n \), where \( m \neq 0 \), we make the following definition.

**Base case** \( \mod \) \( 0 \mod m = 0 \) and

**Step case** \( \mod \)
\[ Sn \mod m = \begin{cases} 0 & \text{if} \quad S(n \mod m) = m \\ S(n \mod m) & \text{else} \end{cases} \]

**Base case** \( \div \) \( 0 \div m = 0 \).

**Step case** \( \div \)
\[ Sn \div m = \begin{cases} S(n \div m) & \text{if} \quad (Sn \mod m) = 0 \\ n \div m & \text{else} \end{cases} \]
Again there is a reason we recurse over the first argument: The result of dividing $S_n$ by $m$ is closely related to that of dividing $n$ by $m$, but the result of dividing $n$ by $S_m$ is not close to that of dividing $n$ by $m$. Working with these definitions is quite delicate and proofs of simple statements are quite involved. Note that giving a definition by cases may not be what one would like, and you are asked to find an alternative in Exercise 133.

**Example 6.39.** Note that in the preceding example we are assuming that we know when two elements of $\mathbb{N}$ are equal. Strictly speaking that is something we have not yet defined. We do this by thinking of it as a function which takes two arguments from $\mathbb{N}$ and returns $0$ or $S_0 = 1$. We want to give a recursive definition of a function which for input $m$ and $n$ gives $1$ if and only if $m = n$, and $0$ else. This can be done as follows (compare Example 6.13).

**Base case** $f_\equiv$. $f_\equiv(0, 0) = S0$.

**Step case** $f_\equiv:m, 0$. $f_\equiv(Sm, 0) = 0$.

**Step case** $f_\equiv:0, n$. $f_\equiv(0, Sn) = 0$.

**Step case** $f_\equiv:m, n$. $f_\equiv(Sm, Sn) = f_\equiv(m, n)$.

Here you can see how recursion over two arguments works, but note that this is a particularly simple case. The code from Code Example 6.13 can be adjusted to implement this function for the class Nat. Note that we have not said which source and target is intended for this function. Clearly the source is meant to be $\mathbb{N} \times \mathbb{N}$, but there are at least two sensible target sets: One might pick $\{0, 1\}$, or one might pick $\mathbb{N}$ again. It is the latter option that you are expected to use in Exercise 133.

**Exercise 132.** Similar to the definition of $f_\equiv$ above give a definition of $f_\leq$, with $f_\leq(m, n) = 1$ if and only if $m \leq n$.

It is possible to combine $f_\equiv$ with the previously defined operations $\div$ and $\cdot$ to give a definition of $\text{mod}$ which does not require the use of a definition by cases, see the following exercise.

**EExercise 133.** Justifying each step show the following statements by induction.

(a) $0 \div n = 0$ for all $n \in \mathbb{N}$.
(b) $P(Sn) = n$ for all $n \in \mathbb{N}$.
(c) $P(Sn \div m) = n \div m$ for all $m, n \in \mathbb{N}$.
(d) $n \div n = 0$ for all $n \in \mathbb{N}$.

\[^{34}\text{Note that a special case of this operation appears in Exercise 130.}\]
(e) Give a definition of mod which does not use a case distinction, but which instead uses the function $f_\equiv$ from Example 6.39, viewed as a function from $\mathbb{N}$ to $\mathbb{N}$. Argue that your definition agrees with the one from the notes.

(f) Show that $n \mod S0 = 0$ for all $n \in \mathbb{N}$. Hint: Use the definition of mod from the previous part.

Hint: You may use statements from previous parts even if you have not proved them.

Optional Exercise 27. If you are looking for a more challenging proof, try to show that $n \mod n = 0$ for all $n \in \mathbb{N}$, or that $n \div 1 = n$ for all $n \in \mathbb{N}$.

The material in this section to this point gives us a rigorous definition of the natural numbers, as well as definitions of commonly used operations on them. This may appear somewhat tedious, covering only things you already knew. But mathematics is all about building everything from first principles, and doing so in a rigorous way. One might continue the development sketched above until one has assembled all the information about the operations on the natural numbers that is summarized in Chapter 0, but we don’t have the time for this. Hopefully the above gives you a flavour of how this might work.

### 6.4.3 Advanced operations for natural numbers

From here on we go back to using the usual names for natural numbers, so we write 1 instead of $S0$, $n + 1$ instead of $Sn$, and so on. The step case in an induction proof is now the more familiar step from $n$ to $n + 1$. Note that in some of the examples below we use induction hypotheses which are more complicated than assuming that the given statement holds for some number $n$ and then proving that this implies it holds for $n + 1$. in these cases we make the induction hypothesis explicit.

From here on we also use the usual operations for natural numbers without referring to their formal recursive definitions. But you should still think about the properties you use, and make them explicit in your proofs.

Example 6.40. In *Java: Just in Time* the example given to introduce the idea of a recursively defined function is that of the factorial function (see Section 22.2). We define this operation as follows.

**Base case** $!$.  $0! = 1$.

**Step case** $!$.  $(n + 1)! = (n + 1) \cdot n!$.

You can see sample calculations of this function in *Java: Just in Time*. Turned into Java code (for integers) this looks something like this:

```java
public static int factorial (int n);
```
{  
    if (n==0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}

Note that we are explicitly doing a test:

- If we are in the base case then we return the appropriate number,
- else we know that we are in a case were $n$ is of the form $m + 1$,
  and so it is safe to subtract one from $n$, and stay in $\mathbb{N}$.

Also note that this code makes no attempt to determine whether the argument is a negative number. You may want to think about what happens if it is called for a negative number $n$.

But apart from these minor differences the code is a very straightforward translation of the mathematical definition. Understanding how the mathematics works therefore can help you write recursive programs. You might note that $1! = 1$, and so if $n = 1$ there's really no need to do a recursion step. You can therefore make the code more efficient (but maybe less clear) by writing instead:

```java
public static int factorial (int n)  
{  
    if (n <= 1)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```

When it comes to calculating the result of reasonably complex operations on natural numbers recursion can be a useful tool.

**Euclid’s algorithm**

Given two natural numbers $m$ and $n$ we can find the largest number dividing both of them, their *greatest common divisor*.

**Example 6.41.** The following is known as *Euclid’s algorithm*. It appears in COMP26120 as an example of a recursive algorithm.

Assume that $a \leq b$ are natural numbers. We set

$$r_0 = b \quad \text{and} \quad r_1 = a.$$  

By Fact 2 from Chapter 0 we can find natural numbers $r_2$ and $k_1 < r_1$.
with the property that
\[ b = r_0 = k_1 r_1 + r_2 = k_1 a + r_2. \]

We keep applying this idea and, invoking Fact 2, we define
\[ r_{i+2} \text{ with } 0 \leq r_{i+2} < r_{i+1} \quad \text{and} \quad k_{i+1} \]
to be the unique numbers with the property that
\[ r_i = k_{i+1} r_{i+1} + r_{i+2}. \]
In other words, we have
\[ k_{i+1} = r_i \div r_{i+1} \quad \text{and} \quad r_{i+2} = r_i \mod r_{i+1}. \]

Note that we have that
\[ r_0 > r_1 > \cdots \]
Any strictly descending sequence of natural numbers must be finite. We may apply Fact 2 and construct new elements for the sequence until we get the number 0, let’s say when we reach \( r_{n+1} \), so that
\[ r_{n-1} = k_n r_n + 0. \]

The number \( r_n \) plays a particular role for \( a \) and \( b \). We claim that \( r_n \) is the greatest common divisor of \( a \) and \( b \). More specifically we show that
- the number \( r_n \) divides both, \( a \) and \( b \) and
- if \( c \) divides both, \( a \) and \( b \) then \( c \) divides \( r_n \).

These proofs are carried out by induction. To prove the first claim we show that
\[ \text{for all } 0 \leq i \leq n \quad r_n \text{ divides } r_i. \]
This means that in particular, \( r_n \) divides both, \( r_1 = a \) and \( r_0 = b \).

Note that this proof is a bit different from other induction proofs we have seen in that
- It proceeds backwards from \( n \) down to 0.
- It requires two base cases since the step case requires that the claim holds for the two previously considered numbers.

For this reason we state the induction hypothesis explicitly.

**Base case** \( n \). Obviously \( r_n \) divides \( r_n \).

**Base case** \( n - 1 \). Since \( r_{n-1} = k_n r_n \) we have that \( r_n \) divides \( r_{n-1} \).

**Ind hyp.** We know that \( r_n \) divides \( r_{i+1} \) and \( r_{i+2} \).
Step case. We show that \( r_n \) divides \( r_i \). By construction we have

\[
r_i = k_{i+1}r_{i+1} + r_{i+2}.
\]

Since by induction hypothesis we know that

\[
r_n \text{ divides } r_{i+1} \quad \text{ and } \quad r_n \text{ divides } r_{i+2}
\]

we can find natural numbers \( l_{i+1} \) and \( l_{i+2} \) with the property that

\[
r_{i+1} = l_{i+1}r_n \quad \text{ and } \quad r_{i+2} = l_{i+2}r_n,
\]

and so

\[
r_i = k_{i+1}r_{i+1} + r_{i+2}
\]

\[= k_{i+1}l_{i+1}r_n + l_{i+2}r_n
\]

\[= (k_{i+1}l_{i+1} + l_{i+2})r_n,
\]

which means that

\[
r_n \text{ divides } r_i.
\]

We leave the proof of the second claim as an exercise.

In COMP26120 you will think about the complexity of this algorithm, that is, you will worry about how many \( r_i \) have to be calculated until the greatest common divisor has been reached.

Exercise 134. Show the second claim from the preceding example.

Recursion can be used beyond defining operations on natural numbers.

Example 6.42. We can also use recursion to define subsets of sets already defined. For example, the set of even numbers (compare Examples 0.7 and 0.15) \( M_2 \) can be defined to be the smallest set satisfying the following conditions.

**Base case** \( M_2 \). \( 0 \in M_2 \).

**Step case** \( M_2 \). \( n \in M_2 \) implies \( 2 + n \in M_2 \).

The base case tells us that \( 2 \cdot 0 = 0 \) is in \( S \), and using the step case we get

\[
0 + 2 = 2 = 2 \cdot 1 \in S,
\]

and applying the step case again we get

\[
2 + 2 = 4 = 2 \cdot 2 \in S,
\]

and so on. Note that we do not have to refer to multiplication in this definition—all we have to do is keep adding 2.
Example 6.43. For another example, to define a set $P_2 \subseteq \mathbb{N}$ containing exactly the powers of 2 we can use the following.

**Base case $P_2$.** $1 \in P_2$.

**Step case $P_2$.** $n \in P_2$ implies $2 \cdot n \in P_2$.

The base case tells us that $2^0 = 1$ is in $S$, and using the step case we get

$$1 \cdot 2 = 2 = 2^1 \in S,$$

and applying the step case again we get

$$2 \cdot 2 = 4 = 2^2 \in S,$$

and so on. Note that we do not have to refer to exponentiation in this definition—all we have to do is keep multiplying with 2.

Exercise 135. Give a recursive definition for each of the following sets.

(a) The set of odd natural numbers as a subset of the natural numbers.

(b) The set of all non-empty lists over $\mathbb{N}$ for which the $n$th element (from the right), for $n \geq 2$, is equal to twice the previous element.

(c) The set of all non-empty lists over $\mathbb{N}$ for which the $n$th symbol (from the right), for $n \geq 3$, is equal to the sum of the previous symbols.

(d) The set of full binary trees such that each label of a node that is not a leaf is the sum of the labels of its children.

(e) The set of full binary trees such that each label of a node that is not a leaf is the sum of the leaves below it.

*Hint: You may use previously defined operations. If you aren’t sure what the step case might look like, construct some lists or trees that satisfy the given criteria.*

Example 6.44. We give a definition of the the $\sum$ operator which makes an appearance in Chapter 4. Assume that $N$ is a set of numbers. Assume that $n$ is a natural number and that $a_i \in N$ for $1 \leq i \leq n$, The operation

$$\sum_{i=1}^{n} a_i$$

is recursively defined as follows.

**Base case $\sum$.** $\sum_{i=1}^{0} a_i = 0$.

**Step case $\sum$.** $\sum_{i=1}^{n+1} a_i = (\sum_{i=1}^{n} a_i) + a_{n+1}$. 

For some sums it is possible to find a simplification, and in those cases a proof that this works is usually by induction.

**Example 6.45.** We show that for all \( n \in \mathbb{N} \),

\[
\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.
\]

**Base case** \( \mathbb{N} \). We have

\[
\sum_{i=1}^{0} i = 0 \quad \text{base case } \sum
\]

\[
= 0 \cdot \frac{1}{2} \quad 0 \cdot n = 0 \text{ in } \mathbb{N}.
\]

**Step case** \( \mathbb{N} \). For this case we have

\[
\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n + 1) \quad \text{step case } \sum
\]

\[
= \frac{n(n+1)}{2} + (n + 1) \quad \text{ind hyp}
\]

\[
= \frac{n(n+1)}{2} + \frac{(n+1)^2}{2} \quad 2/2 = 1, 1 \text{ unit for mult}
\]

\[
= \frac{(n+1)(n+2)}{2} \quad \text{distr and comm of mult}
\]

\[
= \frac{(n+1)(n+2)}{2} \quad \text{distr for mult}.
\]

Note that sometimes sums start at 0 instead of at 1, such as in the exercise below. In that case you have

\[
\sum_{i=0}^{n} a_i = \sum_{i=1}^{n+1} a_{i-1},
\]

which is equivalent to the definition below:

**Base case** \( \sum \).

\[
\sum_{i=0}^{0} a_i = a_0
\]

**Step case** \( \sum \).

\[
\sum_{i=0}^{n+1} a_i = (\sum_{i=0}^{n} a_i) + a_{n+1}.
\]

**Exercise 136.** Show the following statements by induction.

(a) For \( r \in \mathbb{R} \setminus \{1\} \) and \( n \in \mathbb{N} \) we have \( \sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1} \).

(b) For \( n \in \mathbb{N} \) we have \( \sum_{i=0}^{n} i(i + 1) = \frac{n(n+1)(n+2)}{3} \).

(c) For \( n \in \mathbb{N} \) we have \( \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \).

(d) For \( n \in \mathbb{N} \) we have \( \sum_{i=0}^{n} \frac{1}{2^i} = \frac{2^{n+1} - 1}{2^n} \).

(e) For \( n \in \mathbb{N} \) we have \( \sum_{i=0}^{n} i!i = (n + 1)! - 1 \).
For $n \in \mathbb{N}$ we have $1 + \sum_{i=0}^{n} 2^i = 2^{n+1}$.

### 6.4.4 Combinatorial rules

Looking back to the formulae for counting combinations that appear in Section 4.1.2 we can see that these can be shown to do what they are supposed to do by using inductive arguments.

#### Selection with return.

Consider a situation where we have $n$ items to randomly pick from, returning them each time. The claim we have from Section 4.1.2 is that there are $n^i$ combinations when drawing $i$ times. Can we use induction to create a formal argument that this is indeed so?

**Base case** $N$. If there is no draw there is one possible outcome.

**Step case** $N$. If there are $i + 1$ draws then by the induction hypothesis there are $n^i$ many outcomes from the first $i$ draws. Each of those has to be combined with the $n$ possible outcomes of the $(i + 1)$th draw, which means there are

$$n^i \cdot n = n^{i+1}$$

many outcomes for $i + 1$ draws.

#### Selection without return

If we have $n$ items to pick from then according to the formula from Section 4.1.2 for $i$ draws without return there are

$$\frac{n!}{(n-i)!}$$

many combinations. Again we want to give a proof.

**Base case** $N$. If there is no draw there is one possible outcome.

**Step case** $N$. If there are $i + 1$ draws then by the induction hypothesis there are

$$\frac{n!}{(n-i)!}$$

many outcomes from the first $i$ draws. Since $i$ items have been removed there are $n - i$ possibilities for the $(i + 1)$th draw, each of which has to be combined with every outcome from the previous draws, giving

$$\frac{n!}{(n-i)!} \cdot (n-i) = \frac{n!}{(n-i-1)!} = \frac{n!}{(n-(i+1))!}$$

many outcomes

#### Unordered selection without return

We leave the remaining case as an exercise.

---

If you prefer you can start with ‘If there is one draw there are $n$ possible outcomes by assumption’.

338
Exercise 137. For unordered selection without return prove by induction that the number of possible outcomes is

\[ \frac{n!}{(n-i)!i!}. \]

6.4.5 Functions given via recursive specifications

When trying to compute the complexity of a recursive algorithm, that is trying to see how the number of steps changes as the problem size grows, one often ends up having to solve a recurrence relation. This means that one has a recursive specification of a function, and one would like to find a simpler description of that function (using known functions).

Example 6.46. Assume we have a program whose complexity we are trying to work out as a function which takes as its input the problem size \( n \), and gives as its output the number of steps (or an approximation thereof) the program will take when given an input of size \( n \). Further assume that by studying the program we have worked out that

**Base case** \( f \).

\[ f_0 = 1, \text{ that is, if the problem size is } 0 \text{ the program takes one step and} \]

**Step case** \( f \).

\[ f(n+1) = kf_n, \text{ for some } k \in \mathbb{N}, \text{ that is, if the problem size grows by one the program needs } k \text{ times the number of steps from before}. \]

Working with this specification is unwieldy. In this case it’s quite easy to give an alternative description of \( f \), but let’s see how one might arrive there. It’s probably easiest to first of all compute the first few values.

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_n )</td>
<td>1</td>
<td>( k )</td>
<td>( k^2 )</td>
<td>( k^3 )</td>
<td>( k^4 )</td>
</tr>
</tbody>
</table>

We might now guess that \( f_n = k^n \), but we really should verify that this fits with the original description. As with most recursive statements the obvious way of proving this is by induction.

**Base case** \( f \).

\[ f_0 = 1 = k^0, \text{ so we have a match}. \]

**Step case** \( f \).

\[ f(n+1) = kf_n \quad \text{definition } f \]

\[ = k \cdot k^n \quad \text{induction hypothesis} \]

\[ = k^{n+1} \quad \text{calcs in } \mathbb{N}, \]

and this gives another match.

Example 6.47. Let’s do a more complicated example.

**Base case** \( g \).

\[ g_0 = 0. \]
Base case $g$: $g_1 = 1$.

Step case $g$: $g(n+2) = 2g(n+1) - gn + 2$.

Note that we have two base cases here. They are required because the step case relies on two previously calculated values. Again we work out the first few values.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$g_0$</th>
<th>$g_1$</th>
<th>$g_2$</th>
<th>$g_3$</th>
<th>$g_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
</tr>
</tbody>
</table>

Once more there’s an obvious guess: We guess that $g_n = n^2$. Again we have to provide a proof. Note that we have to cover two base cases now, and that the induction hypothesis is now: $g_k = k^2$ for all $k < n + 2$. In particular we use below the induction hypothesis that

$$g(n+1) = (n+1)^2$$

and

$$g_n = n^2.$$

Base case $g_0$. $g_0 = 0 = 0^2$.

Base case $g_1$. $g_1 = 1 = 1^2$.

Ind hyp $g$. We have

$$g_n = n^2$$

and

$$g(n+1) = (n+1)^2.$$

Step case $g$.

$$g(n+2) = 2g(n+1) - gn + 2$$

def $g$

$$= 2(n+1)^2 - n^2 + 2$$

induction hypothesis

$$= 2n^2 + 4n + 2 - n^2 + 2$$

calcs in $\mathbb{N}$

$$= n^2 + 4n + 4$$

calcs in $\mathbb{N}$

$$= (n+2)^2$$

calcs in $\mathbb{N}$.

CExercise 138. For the following recursive specifications, work out an alternative representation of the given function and prove by induction that it satisfies the given specification.

(a) Base case $f$. $f_0 = 0$.

Step case $f$. $f(n+1) = fn + 2$.

(b) Base case $f$. $f_0 = 1$.

Step case $f$. $f(n+1) = (n+1)fn$.

(c) Base case $f$. $f_0 = 2$.

Step case $f$. $f(n+1) = (fn)^2$. 
Note that the examples given in these notes have been carefully chosen so that it is possible to guess a closed form for the underlying function. Not all recurrence equations that appear in real-world programs are as easy as this. An example is provided by the following optional exercise. Typically recursive specifications arise when one wants to compute the complexity of a recursive program, and often those occurring in practice are much harder to solve than our examples.

**Optional Exercise 28.** A popular exercise in writing recursive programs is to compute the Fibonacci sequence. Consider the following recursive specification for a function.

Base case $f$:0. \hspace{1cm} f_0 = 0.

Base case $f$:1. \hspace{1cm} f_1 = 1.

Step case $f$. \hspace{1cm} f_{n+2} = f_{n+1} + f_n.

(a) Calculate the first few values of $f$.

(b) Show that for

$$\alpha = \frac{1 + \sqrt{5}}{2} \quad \text{and} \quad \beta = \frac{1 - \sqrt{5}}{2},$$

we have

$$\alpha^{n+2} = \alpha^{n+1} + \alpha^n \quad \text{and} \quad \beta^{n+2} = \beta^{n+1} + \beta^n,$$

so these numbers satisfy the equations defining $f$. 

341
(c) Show that the function

\[
fn = \frac{\alpha^n - \beta^n}{\alpha - \beta}
\]

\[
= \frac{(\frac{1 + \sqrt{5}}{2})^n - (\frac{1 - \sqrt{5}}{2})^n}{\sqrt{5}}
\]

\[
= \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{2^n \sqrt{5}}
\]

satisfies the original condition for \( f \).

Would you have guessed this definition for \( f \)?

**Optional Exercise 29.** Note that it is possible to have recurrence relations with more than one variable. A typical example are the *binomial coefficient*, which can be thought of as given by the recurrence relation

**Base case** binom. \( \binom{n}{0} = 1 \).

**Step case** binom. \( \binom{n+1}{i+1} = \binom{n}{i} + \binom{n}{i+1} \).

Show that this gives the same definition as the more common

\[
\binom{n}{i} = \frac{n!}{i!(n-i)!}.
\]

Note that the recursive definition does not require multiplication, just addition. You may have seen this definition at work in *Pascal’s triangle*.

### 6.5 Further properties with inductive proofs

Many properties involving natural numbers can be shown by induction. The following exercise gives a taste of those.

**Example 6.48.** Here is an inductive proof of a property of complex numbers, but invoking natural numbers where it counts. We show that

\[ |z| = 1 \quad \text{implies} \quad \text{for all } n \in \mathbb{N}, \ |z^n| = 1. \]

**Base case** \( \mathbb{N} \). We have \( |z^0| = |1| = 1 \).

---

36Check out https://en.m.wikipedia.org/wiki/Fibonacci_number to read up on where this sequence originates and some of its interesting properties.
Step case \( \mathbb{N} \). For this case we have

\[
|z^{n+1}| = |z^n| \quad \text{def } (-)^{n+1}
\]
\[
= |z^n||z| \quad |zz'| = |z||z'|, \quad \text{Ex 16}
\]
\[
= 1 \cdot 1 \quad \text{ind hyp, assumptn}
\]
\[
= 1 \quad 1 \text{ unit for mult}
\]

Exercise 139. The following statements can be shown by induction but some creativity is required to see how the proof might work.

(a) Show that every natural number greater than or equal to 12 is the sum of multiples of the numbers 4 and 5. This means that if we have 4p and 5p stamps then they can be combined to get any amount from 12p up. Hint: Work out solutions for smaller numbers, up to 24 or so. A pattern should emerge. You will require a number of case distinctions.

(b) Show that for all natural numbers \( n \neq 0 \) we have that \( 7^n - 1 \) is divisible by 6, as is \( n^3 - n \).

(c) Assume that \( n \in \mathbb{N} \) and \( n \geq 2 \). Show that a set of size \( n \) has \( n(n-1)/2 \) two-element subsets. Hint: This is an exercise that wants to be solved by reasoning in English, rather than manipulating some formula.

After all this you may wonder what ‘recursion theory’ is all about. It is the study of those functions \( \mathbb{N} \rightarrow \mathbb{N} \) which can be defined using recursion. This is more involved than you may think—because one may use previously defined functions to define more sophisticated ones the functions that can arise in this way are considerably more complicated than those that appear in this section.

6.6 More on induction

The examples given above are comparatively simple cases of proofs by induction. We briefly discuss situations where a proof becomes more complicated.

- The induction hypothesis in most of our examples is very straightforward in the sense that for each step case, we merely require the statement for each ‘ingredient’. For example, in the case of a propositional formula, assuming the statement holds for the formula \( A \) and the formula \( B \) we show that it holds for \( A \land B \). Sometimes one has to require the statement for ‘all ingredients built so far’. In the above example this would mean assuming the statement for \( A, B \), and all their subformulæ.

- In some cases it is not sufficient to assume the given claim for all entities built so far’. Instead one finds that when trying to prove the step case that one needs a strong statement than the one that is at issue. This often gives some idea of an alternative, stronger statement.
that can be shown by induction. This is sometimes called *strengthening the induction hypothesis*. We have not seen any examples of this.

- It is perfectly possible to *nest* induction proofs, that is, inside a proof by induction one requires another proof, also by induction, of an unrelated statement. We have not seen examples of this.

- If one tries to show a statement which contains more than one variable one of the following cases will apply if the statement is provable by induction at all:
  
  - We may prove the statement by induction over one of the variables, treating the others as parameters. Examples of this are Examples 6.11 and 6.33.
  
  - We may prove the statement by giving an induction proof for one of the variables, which inside contains induction proofs for the other variables (where one has to be careful to state the various induction hypotheses carefully to ensure that they do not assume anything unwarranted). In the case where there are two variables this is known as *double induction*. We have not seen examples of this.
Chapter 7

Relations

So far when trying to connect two sets we have only looked at functions. This assumes that we have a mechanism for turning elements from the source set into elements of the target set. However, there are other ways of making connections.

Example 7.1. In a database we typically want to think of tables, and indeed people often talk about ‘relational databases’. These can be viewed as relations of a general kind. For a simple example, think of a library which has a table of members (uniquely determined by their membership numbers), a table of books (uniquely determined by catalogue numbers), and a table which keeps track of which book is currently on loan to which member. We can think of this as connecting the set of all members $M$ (represented by the set of all valid membership numbers) to the set of all books $B$ (represented by the set of all current catalogue numbers). This connection cannot be thought of as a function: A given member may have no, or several, books on loan, so we cannot produce a unique output for each input. Instead we can think of this table as a subset $L$ of the product $M \times B$ with the property that

$$(m, b) \in L \quad \text{if and only if} \quad \text{member } m \text{ currently has book } b \text{ on loan.}$$

This is an example of a relation from $M$ to $B$.

In terms of a database one would describe this as a relation schema, and would give a type for it along the following lines:

$$\text{OnLoan(member:int,book:int)},$$

assuming that membership and catalogue numbers are implemented as integers. The entries in the database are exactly the members of the relation.

In these notes we largely restrict ourselves to relations connecting two sets, which means relation schema with two entries. Databases often have relation schemas with more entries, for example a database that keeps track of members of the university might have a relation schema including

$$(\text{title, name, building, office number, phone number}).$$
The general ideas regarding relations that are introduced below apply to this kind of situation as well.

7.1 General relations

We use relations all the time, even though you may not have been aware of that.

It is often convenient to think of relations as generalizations of functions, and sometimes a similar notation is used.

A relation \( R \) from a set \( S \) to a set \( T \) is given by a subset of \( S \times T \). This is sometimes written as

\[
R : S \rightarrow T,
\]

but this is not universal.

A function is a special kind of relation: Given a function \( f : S \rightarrow T \) we have its graph

\[
\{(s, fs) \in S \times T \mid s \in S\},
\]

which is a subset of \( S \times T \). It is standard to identify \( f \) with its graph to view it as a relation. Proposition 2.1 tells us which relations are the graphs of functions, namely those relations \( R \subseteq S \times T \) where for every \( s \in S \) there is a unique \( t \in T \) such that \((s, t) \in R\).

Examples of relations are abundant.

**Example 7.2.** We give a few examples of relations from a number of areas.

(a) The relation from the set of students in the School to the set of academics where a student is related to an academic if during the current academic year the student is enrolled on a course unit on which the academic teaches.

(b) The relation from the set of students in the School to the set of COMP course units offered which relates a student to all the course units he or she is enrolled on.

(c) The relation from the set of real numbers to the set of real numbers where \( x \) is related to \( y \) if \( x = y^2 \).

(d) The relation between Java programs where one program is related to another if they can be viewed as computing the same thing.\(^1\)

(e) The relation of equality for a number of entities, for example, equality of numbers, equality of fractions, and more generally the equality of ‘arithmetic expressions’, that is, expressions written using numbers and operations such as addition, multiplication and inverses with respect to these. We usually consider two such expressions to be equivalent (or equal) if they evaluate to the same number.

---

\(^1\)Defining this in general in a rigorous way is non-trivial, but it is fairly easy if one only looks at Java programs which can be thought of as having a natural number as input and a natural number as output.
We can picture relations between small finite sets in a picture, similar to how we draw functions between such sets.

**Example 7.3.** We show how to picture a small relation between two different sets. For small relations on the same set see Section 7.2.2.

This relation goes from the set \{a, b, c\} to the set \{1, 2, 3, 4\}. It relates

- a to 1, 3 and 4,
- b to 3, and
- c to no element.

We typically write this relation as the collection of pairs from the set

\[ \{a, b, c\} \times \{1, 2, 3, 4\} \]

which it contains, which in the case of the example above is

\[ \{(a, 1), (a, 3), (a, 4), (b, 3)\}. \]

### 7.1.1 Important notions

There are two common notations for relations. One is to rely on the idea that a relation \( R \) from \( S \) to \( T \) is a subset of \( S \times T \) and so to denote the fact that \( R \) relates an element \( s \) of \( S \) to an element \( t \) of \( T \) by

\[ (s, t) \in R. \]

Sometimes infix notation is preferred, and instead of \( (s, t) \in R \) one might write

\[ s \sim R t. \]

Where infix notation is used it is not unusual to see symbols, rather than letters, to denote a relation; for example you may find a relation \( \sim \) from \( S \) to \( T \) where the same fact is written as

\[ s \sim t. \]

**Example 7.4.** An example of a relation that is typically written in this manner this is *equality of arithmetic expressions*. To denote that the two expressions \( 2/4 \) and \( 1/2 \) denote the same number in \( \mathbb{Q} \) or \( \mathbb{R} \) we write

\[ 2/4 = 1/2; \]
the equal symbol $=$ being written in infix notation.

**Example 7.5.** Another example is the notion of *semantic equivalence* of propositions from the material on logic, where the notation

$$A \equiv B$$

is used to denote the fact that with respect to every valuation $A$ has the same boolean interpretation as $B$.

Both notations are routinely used and you should become comfortable with both.

Each set has a special relation: Given a set $S$ the **identity relation** $I_S$ on $S$ is given by

$$(s,s) \in S \times S \mid s \in S.$$  

In other words, every element is related to itself, and to nothing else. The identity relation on $S$ is the graph of the identity function $\text{id}_S$ on $S$.

Given a relation $R$ from $S$ to $T$ there is an easy way of turning it into a relation from $T$ to $S$: The **opposite relation** $R^{\text{op}}$ of $R$ is given by

$$R^{\text{op}} = \{(t,s) \in T \times S \mid (s,t) \in R\}.$$  

In other words we ‘turn the relation around’ by changing the order of the pairs, and also switch the ‘source’ and ‘target’.

**Example 7.6.** For the relation given in Example 7.3 above the opposite relation is given by the following picture.

![Diagram](image)

**Example 7.7.** Consider the relation ‘is a child of’ as a relation on the product with itself of the set of all people, where

$$(a,b)$$

being an element of the relation means that $a$ is a child of $b$. The opposite of the ‘is a child of’ relation is the ‘is a parent of’ relation, where

$$(a,b)$$

is in the relation if and only if $a$ is a parent of $b$.

Note that since relations are sets we can apply set operations to them. In particular, given a relation $R$ from $S$ to $T$ there is its complement,

$$(S \times T) \setminus R,$$
and given two relations \( R \) and \( R' \) from \( S \) to \( T \) we may form

\[
R \cup R' \quad \text{and} \quad R \cap R'.
\]

Relations are a little like functions in that one can define their composition. Let \( R \) be a relation from \( S \) to \( S' \), and let \( R' \) be a relation from \( S' \) to \( S'' \). The \textbf{relational composite} \( R ; R' \) is given by

\[
R ; R' = \{(s, s'') \in S \times S'' \mid \exists s' \in S' \text{ such that } ((s, s') \in R \text{ and } (s', s'') \in R')\}.
\]

**Example 7.8.** Consider the two relations given by the following picture.

If we ‘overlay’ the two we can more easily see what the composite of the two relations is.

The composite connects

- a node in the left-most set with
- a node in the right-most set if and only if
- they are connected by a line through the set in the middle (that is a red line followed by a blue one).

**Example 7.9.** Consider the ‘is a child of’ relation from Example 7.7. We may form the relational composite of this relation with itself, and the result is the ‘is a grandchild of’ relation.

**Example 7.10.** Assume we have two relation schema (compare Example 7.1) in a database,

- one connecting members (given by their membership number) with borrowed books (given by their catalogue number) and
- one connecting books (given by catalogue number) and their titles.
If one wants to create a new relation scheme that connects members (given by their membership number) with the titles of the books they have on loan one has to form the relational composite of the two underlying relations.

On the left we have a table describing the relations between members and the catalogue numbers of the books they have on loan, and on the right a table describing the title of the book corresponding to a catalogue number.

<table>
<thead>
<tr>
<th>Member 1</th>
<th>00002</th>
<th>00003</th>
<th>00005</th>
<th>00007</th>
<th>00011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Member 3</td>
<td>00002</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Member 3</td>
<td></td>
<td>00003</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Member 3</td>
<td>00007</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Member 3</td>
<td></td>
<td></td>
<td>00005</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Member 3</td>
<td></td>
<td></td>
<td></td>
<td>00007</td>
<td></td>
</tr>
<tr>
<td>Member 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00011</td>
</tr>
</tbody>
</table>

The relational composite of the underlying relations, written again as a table, is the following:

<table>
<thead>
<tr>
<th>Member 1</th>
<th>The Joys of Java</th>
<th>Maths for Dummies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member 1</td>
<td>Maths for Dummies</td>
<td></td>
</tr>
<tr>
<td>Member 3</td>
<td>Higher Category Theory</td>
<td>Topoi and Theories</td>
</tr>
<tr>
<td>Member 3</td>
<td>Topoi and Theories</td>
<td></td>
</tr>
<tr>
<td>Member 3</td>
<td>Multiversal Algebra</td>
<td></td>
</tr>
</tbody>
</table>

But note that this is a special example, which makes it look as if every entry in the first table gives rise to an entry in the resulting table. This is the case because our second table defines a very special relation, namely one that is functional: For every catalogue number there exists exactly one book title (namely the title of the corresponding book). Moreover, since every catalogue number can be on loan to at most one person, the first relation is also special.

**Example 7.11.** Assume we have a different database. We have a relation schema connecting applicants to the school with their computer science interests, and another relation schema which suggests related interests.

<table>
<thead>
<tr>
<th>Wong</th>
<th>AI</th>
<th>AI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wong</td>
<td>HCI</td>
<td>AI</td>
</tr>
<tr>
<td>Kim</td>
<td>Maths</td>
<td>Security</td>
</tr>
<tr>
<td>Anna</td>
<td>AI</td>
<td>Maths</td>
</tr>
<tr>
<td>Anna</td>
<td>Hardware</td>
<td>Maths</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Probabilities</td>
</tr>
</tbody>
</table>

The relational composite of the underlying relations, written again as a table, is the following:
Example 7.12. In COMP11212 the notion of a *simulation* between finite state automata is introduced. This is a relation that connects the states of the two automata. Only relations with particular properties are valid simulations. This is an example where relations specifically appear in a computer science context.

Optional Exercise 30. Show that if you have a relation $R$ which is a simulation from $A$ to $A'$, and if $R'$ is a simulation from $A$ to $A''$ then their composite as relations, $R; R'$, is a simulation from $A$ to $A''$.

Exercise 140. Show the following for relational composition:

(a) Given a relation $R$ from $S$ to $T$, show that for the identity relation $I_S$ on $S$ we have $I_S; R = R$ and similarly that for the identity relation $I_T$ on $T$ we have $R; I_T = R$.

(b) Assume that $R, R'$ and $R''$ are relations that can be composed. Show that $(R; R') ; R'' = R ; (R' ; R'')$.

We begin by looking at a generalization of the notion of a function, and then move to other cases of relations with particular properties.

### 7.2 Partial functions

Sometimes we would like to consider assignments that behave like functions, but which are not defined on the whole source set.

Example 7.13. Division is such a function from

$$\mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q}.$$  

It is defined everywhere with the exception of the subset

$$\mathbb{Q} \times \{0\} \text{ of } \mathbb{Q} \times \mathbb{Q}.$$  

This gives us the choice of defining division
• as a function with source and target
\[ \mathbb{Q} \times (\mathbb{Q} \setminus \{0\}) \rightarrow \mathbb{Q} \]

• or as a partial function
\[ \mathbb{Q} \times \mathbb{Q} \longrightarrow \mathbb{Q} , \]
which is undefined for all those pairs whose second component is 0, that is, pairs of the form \((q, 0)\).

In this example there is a choice regarding what to do, but it can be difficult, or even impossible, to calculate the domain where a particular partial function is defined. Examples of this appear in COMP11212.

Example 7.14. A more interesting example is that of subtraction for the natural numbers. We may define a partial function
\[ \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \]
which maps
\[ (n, m) \longmapsto \begin{cases} 
  n - m & n \geq m \\
  \text{undefined} & \text{else.}
\end{cases} \]

Alternatively a total function would have be\(^2\)
from \(\{(n, m) \in \mathbb{N} \times \mathbb{N} \mid n \geq m\}\) to \(\mathbb{N}\).

Example 7.15. Nothing stops us from defining, for example, for natural numbers \(n\),
\[ fn = \begin{cases} 
  n & \text{the no of atoms in the universe is divisible by } n \\
  \text{undefined} & \text{else}
\end{cases} \]
Calculating where this function is defined is impossible (beyond repeating the definition). If we use this function together with a recursive definition we can define

Base case \(g\). \hspace{1em} g0 = 0.

Step case \(g\).
\[ g(n + 1) = \begin{cases} 
  gn + f(n + 1) & gn, f(n + 1) \text{ both defined} \\
  fn & f(n + 1) \text{ defined, } gn \text{ undefined} \\
  \text{undefined} & \text{else.}
\end{cases} \]

Here it is even harder to work out whether \(g\) is defined for a given \(n\), and there is no simple predicate which tells us whether \(gn\) is defined for a given \(n\).

\(^2\)Or we would have to turn it into a total function such as \(–\) from the previous section.
Example 7.16. In COMP11212 you will study the idea of a *partially decidable partial function*. This is a partial function for which we can find an algorithm to tell us where it is defined, but we demand the termination of the algorithm only in the case where the given partial function is defined at the given element. Specifically, the question is whether there is a While program which, for input $n$, will terminate and give the answer 1 if the function is defined at $n$. Such a program is known as a *partial decision procedure*.

In a computer science context a typical application of this idea is to have a program $P$ that takes some input, for example a natural number, and produces an output, say another natural number. We may then define a partial function which assigns

$$n \rightarrow \begin{cases} \text{output of } P \text{ on input } n & \text{if } P \text{ produces output for input } n \\ \text{undefined} & \text{else.} \end{cases}$$

The reason why $P$ might not produce an output for the given input could be that it attempts a division by 0, or that it runs into an infinite loop on certain inputs. You will study this idea in more detail in COMP11212.

We give a formal definition for the concept used in the examples above.

**Definition 50: partial function**

Let $S$ and $T$ be sets. A *partial function from $S$ to $T$* is an assignment where

for every $s \in S$ there is at most one $t \in T$ with $s \rightarrow t$.

We use a different kind of arrow, one with only ‘half a tip’

$$f : S \rightarrow T$$

to indicate that the function described is partial.

When we have partial functions between small sets we can draw pictures of partial functions, similar to those used for (total) functions in Section 0.3. Here for every element of the source set we may have at most one element of the target set which it is connected with.

If $f$ is a partial function then when we write $fs$ we cannot be sure whether this defines an element of $T$, or whether this is undefined. This can be awkward when one tries to argue about partial functions. For this reason it is fairly customary to use an extra symbol\(^3\) $\perp$.

---

\(^3\)This looks like the symbol we used in Chapter 3 for propositions whose boolean interpretation relative to every valuation is 0, and that is no coincidence. People usually pronounce it ‘bottom’—see Definition 62 for the reason why.
If $f: S \rightarrow T$ is a partial function then we write

$$fs = \bot$$

in the case\textsuperscript{1} where $f$ is not defined at $s$.

**Example 7.17.** Every function from $S$ to $T$ is also a partial function from $S$ to $T$.

When both, proper and partial functions are around people sometimes talk of **total functions** to distinguish proper functions, which are defined everywhere, from partial ones.

**Example 7.18.** Recall the notion of a list over a set $S$ from the previous chapter. We may want to define a function that returns the most recently added element of the list (if it exists). This should be a partial function head from the set of lists over $S$ to $S$. It is partial because it is undefined for the empty list. There is a recursive definition which says

$$\text{head}(s : l) = s.$$  

By not providing a definition for head at $[]$ we are implicitly stating that this function is not defined for the empty list, which means that it is partial. There really is no sensible way of extending this function to the empty list—how would we pick an element of $S$ to return?

It is for reasons such as the function in this example (and the fact that when programming, illegal operations such as division by 0 cannot be prevented from happening in some automated way) that computer scientists have to consider partial functions.

Above we discussed the idea that we can make a partial function total by restricting its source to include only those elements for which the function is defined. This idea has a name.

The **domain of definition of a partial function** $f: S \rightarrow T$ is defined as

$$\text{dom } f = \{ s \in S \mid fs \text{ is defined} \}.$$  

**Proposition 7.1**

For every partial function $f: S \rightarrow T$ there is a unique total function

$$g: \text{dom } f \rightarrow T$$

with the property that for all $s \in \text{dom } f$ we have

$$gs = fs.$$  

\textsuperscript{4}This only works if $\bot$ is not an element of the target set $T$.  

354
Proof. We can use the desired equality as a definition for $g$. This clearly gives a total function with the required source and target and the condition ensures that for every element of $\text{dom } f$ there is a unique element of $T$ to which to map it.

This means it is possible to work with total functions if we prefer to do so. However in theoretical computer science there exist a number of concepts around computability theory (and also in recursion theory in mathematics) which are more easily explained using partial functions. Examples occur in COMP11212.

The previous result has a counterpart where we move from total to partial functions.

**Proposition 7.2**

If $f: S \rightarrow T$ is a (total) function from $S$ to $T$ and $S'$ is a superset of $S$ then there is a unique partial function $g: S' \rightarrow T$

with the property that

$\text{dom } g = S$ and $\forall s \in \text{dom } g. gs = fs$.

**Exercise 141.** Prove Proposition 7.2.

Partial functions can be composed just as the case for total functions, but we have to be careful about for which arguments the corresponding composite function is defined.

The **composite of partial functions** $f: S \rightarrow T$ and $g: T \rightarrow U$ is defined as:

$$(g \circ f)_s = \begin{cases} g(fs) & \text{if } f(s) \text{ and } g(f(s)) \text{ are both defined} \\ \perp & \text{else.} \end{cases}$$

Alternatively one may say that $(g \circ f)_s$ is defined if and only if $s$ is in the domain of definition of $f$ and $fs$ is in the domain of definition of $g$.

Note that this subsumes the definition of the composition of functions (defined in Section 0.3).

**Example 7.19.** We illustrate the notion of a composite of two partial functions by using a small example which can be described using a picture.

If we ‘overlay’ the two pictures we can more easily see what the composite has to look like.
The composite of the two partial functions can be drawn by the following procedure:

- Pick an element of the left hand set. If there is an outgoing arrow, follow that arrow. If there is an outgoing arrow from that element, follow that arrow and connect the original element with the resulting element.

- If at any point in the above procedure there is no outgoing arrow for one of the elements then the original element is not connected to anything.

CExercise 142. Consider the following partial functions whose source is \texttt{FBTrees}.

(a) Give a recursive definition of a (possibly partial) function that takes a full binary tree and returns the label of the root of the tree. What is the domain of definition of your function? \textit{Note that this is not a truly recursive function in the sense that when you define the step case you do not have to use the result of the function for the left and right subtrees.}

(b) Give a recursive definition of a (possibly partial) function that takes a full binary tree and returns its left subtree. What is the domain of definition of your function?

(c) What happens if you apply your second function, and then the first, to a tree? Describe the domain of definition of this composite.

EExercise 143. Let \( f: S \rightarrow T \) and \( g: T \rightarrow U \) be partial functions.

(a) Show that the domain of definition of \( g \circ f \) is a subset of the domain of definition of \( f \), that is

\[ \text{dom}(g \circ f) \subseteq \text{dom } f. \]

(b) Give a description of the domain of definition of \( g \circ f \) as a subset of \( \text{dom } f \), without mentioning \( g \circ f \), nor the expression \( g(f(s)) \) for some \( s \in S \).

\textit{Hint: You may use dom } f \text{ and dom } g.
The graph of a partial function $f : S \to T$ is very similar to that of a (total) function (compare page 48). Its definition is

$$\{(s, fs) \in S \times T \mid f \text{ is defined at } s\}.$$

We can characterize those subsets of $S \times T$ which appear as the graph of a partial function (compare Proposition 2.1).

**Proposition 7.3**

A relation $R \subseteq S \times T$ is the graph of a partial function from $S$ to $T$ if and only if

for all $s \in S$ there exists at most one $t \in T$ with $(s, t) \in R$.

**Proof.** It is possible to make minor changes to the proof of Proposition 2.1 to obtain a proof of this result.

In the same way that we can view a function as a relation by considering its graph we may view a partial function that way, and this proposition tells us which relations are the graphs of partial functions.

**Exercise 144.** Show that for partial functions

$$f : S \to T \quad \text{and} \quad g : T \to U$$

we have that

$$\text{gr}(g \circ f) = \text{gr} f \cdot \text{gr} g,$$

where $\cdot$ is the relational composite defined on page 349, and $\text{gr}$ applied to a function gives its graph. Note that this means in particular that composition of (total) functions is subsumed by relational composition.

### 7.2.1 Binary relations

For the remainder of this chapter we only consider relations from one set to itself. Instead of saying that $R$ is a relation $S \to S$ one typically says that $R$ is a **(binary) relation on** $S$. Often it is convenient to drop the ‘binary’ in this case.

**Example 7.20.** Typical examples of binary relations on a set are the following.

(a) Sharing some property, such as having the same size, the same colour, the same nationality, speaking the same language, having the same digits after the decimal point, having a common divisor, evaluating to the same number,

(b) Having the same value under some function (some of the examples from above can also be viewed from this perspective, for example, one could map people to their height, or a real number to its sequence of digits after the decimal point, but one cannot do this for all these examples since a person may have more than one nationality or speak more
than one language),

(c) Other kinds of connections between two members of the same set, for example one person having the contact details for another, or a number being less than or equal to another, or two subsets of a given set being included in each other.

(d) If we look at two objects of class List then we think of them as implementing the same list if the method equal (see Example 6.13) returns true. This defines a binary relation on objects of this class.

A different way of thinking of the same relation is as follows: Every object of class List can be thought of as implementing an element of Lists$S$ in the obvious way. This defines a function

$$\text{Objects of class List} \rightarrow \text{Lists}_S,$$

and two such objects are in the relation if and only if this function maps them to the same list.$^5$

Note that for binary relations the ‘source’ and ‘target’ are identical, which means we may always form the relational composite of a binary relation with itself.

### 7.2.2 Picturing binary relations

If we have a binary relation on a small finite set $S$ it is possible to draw$^6$ it in the form of a directed graph.

**Definition 51: directed graph**

A directed graph on a set $S$ is given$^7$ by a binary relation on $S$. In this context the elements of $S$ are often called the nodes of the graph$^8$ and the pairs in the relation the edges.

**Example 7.21.** Such graphs are often drawn as pictures when the set $S$ is small. Consider the following relation on the set $\{a, b, c, d, e\}$:

$$\{(a, b), (b, a), (b, c), (c, c), (c, d), (d, e), (e, a), (b, d)\}.$$ 

Its picture as a binary relation is as follows.

---

$^5$Compare Example 6.13.

$^6$Above we show how to generally draw relations from a small set $S$ to a small set $T$, but this method makes use of the fact that the relation goes from a set to the same set.

$^7$Some definitions of directed graph exclude connections between an element of $S$ and itself. In that case the graph is given by a collection of pairs of the form $(s, s')$ where $s$ and $s'$ are distinct elements of $S$. When that definition is used, our directed graphs become directed graphs with loops.

$^8$Some people call the nodes of the graph the vertices.
Exercise 145. Draw the following binary relations.

(a) The powerset $\mathcal{P}\{a, b, c\}$ with the relation which relates a subset $S$ of $\{a, b, c\}$ to the subset $S'$ if and only if $S \subseteq S'$.

(b) The set $\{0, 1, 2, 3, 4, 5, 6\}$ with the relation which relates an element $m$ to an element $n$ if and only if they both leave the same remainder when divided by 3, that is, $m \mod 3 = n \mod 3$.

(c) The set $\{0, 1, 2, 3, 4, 5, 6\}$, with the relation which relates an element $m$ to an element $n$ if and only if $m \leq n$.

(d) The set $\{0, 1, 2, 3, 4, 5, 6\}$ with the relation which relates an element $m$ to an element $n$ if and only if $m$ divides $n$.

(e) The compulsory course units for a student on your programme of studies, where one unit is related to another if it is a prerequisite for that unit.

(f) The students in your tutorial group where one student is related to another if they have the same gender.

The identity relation on a set can be pictured easily using these ideas. It is the relation which connects every element of the set with itself, and nothing else.

Example 7.22. For the set $\{a, b, c, d, e\}$ the identity relation is pictured as follows.
When we have a binary relation we picture its opposite slightly differently from the way illustrated in Example 7.6.

**Example 7.23.** For the relation on the set \{a, b, c, d, e\} from Example 7.21, \{(a, b), (b, a), (b, c), (c, c), (c, d), (d, e), (e, a), (b, d)\}, we show both, the given relation and its opposite.

The opposite relation, described as a set of pairs, is \{(b, a), (a, b), (c, b), (c, c), (d, c), (e, d), (a, e), (d, b)\},

We can see how the opposite relation arises from the original by turning around all the arrows—which means that for loops, or for arrows with tips on both ends, nothing changes.

**Further relations on a set**

Very occasionally one may want to connect more than two elements of the same set, in which case one may speak of a

- ternary relation on \(S\) (a subset of \(S \times S \times S\)),
- a quarternary relation on \(S\) (a subset of \(S \times S \times S \times S\)),
- or, more generally, an \(n\)-ary relation on \(S\) (a subset of the \(n\)-fold product of \(S\)).

**7.3 Equivalence relations**

Sometimes when we look at all the elements of a set we don’t necessarily wish to distinguish all the elements of the set. When you are building a specific structure from building blocks, in order to complete the design you
don’t have to consider the colour of a given block, just its shape. So from the point of view of the design, all blocks of the same shape are equivalent.¹ In COMP11212 you have met the notion of a bisimulation between two finite state automata. You can think of a bisimulation as a way of demonstrating that two given automata are equivalent, as far as the language they define is concerned.

More generally whenever we want to think of various entities as equivalent we have to make sure we do this in a safe way. Relations that allow us to do this are known as equivalence relations. These are relations satisfying a number of properties, and we introduce these properties one at a time.

### 7.3.1 Reflexivity

**Definition 52: reflexive**

A binary relation $R$ on a set $S$ is reflexive if and only if it is the case that we have

\[
\text{for all } s \in S \quad (s, s) \in R.
\]

The first order logic formula that describes reflexivity of $R$, where $R$ is a binary relation symbol, is

\[
\forall x. R(x, x).
\]

We may express reflexivity in a very brief way by noting that it means that

\[
I_S \subseteq R,
\]

where $I_S$ is the identity relation on the underlying set $S$.

This means that if our relation is reflexive then every element of the underlying set is related to itself. If we can draw a picture of the relation we can check whether the relation is reflexive by checking that every element has a connection from itself to itself, usually drawn as a little loop.

Typical examples are relations involving some kind of equality.

**Example 7.24.** We provide examples and non-examples for this concept.

(a) Consider the relation between first year students in the School where two students are related if and only if they are in the same tutorial group. Since every student is in the same tutorial group as him- or herself, this relation is reflexive.

(b) Look at the relation between two members of the human population where two people are related if and only if they have the same height in centimetres. Since everybody has the same height as him- or herself, this relation is reflexive.

(c) The relation between two non-zero natural numbers where $m$ is related to $n$ if and only if $m$ divides $n$ is reflexive since every natural number other than 0 divides itself.

¹But, of course, for aesthetic reasons, or to match an existing edifice, you may want to consider the colour when building a structure.
(d) We define the relation between first year students in the School where student $A$ is related to student $B$ if $A$’s student id number is below that of student $B$. This relation is not reflexive since a number is not below itself.

(e) Consider the relation between members of the human population which relates person $A$ to person $B$ if and only if they are siblings. This relation is not reflexive since nobody is their own sibling. On the other hand, relating two people if and only if they have a parent in common is a reflexive relation.

(f) Consider the relation between two natural numbers where $m$ is related to $n$ if $n = 2m$. Since, for example $1 \neq 2 \cdot 1$ this relation is not reflexive.

(g) The relation between elements of $\text{Lists}_S$ which relates two lists if and only if they have the same number of elements is reflexive.

(h) The relation which relates two objects of class $\text{List}$ if the method $\text{equal}$ returns true (see Example 6.13) is reflexive.

(i) The relation of semantic equivalence for propositions is reflexive.

**Exercise 146.** Which of the following relations are reflexive? Justify your answer.

(a) The relation where two first year students in the School are related if their last name starts with the same letter.

(b) Two first year students in the School being related if there is a university society they both belong to.

(c) The following relation on the set $\{a, b, c, d\}$:

$$\{(a, a), (b, c), (c, b), (b, b), (c, d), (d, c), (d, d)\}$$

(d) The relation on $\mathbb{N}$ where two numbers are related if and only if they have a common divisor greater than 1.

It is easy to make a relation reflexive in a unique and minimal way. Given a binary relation $R$ on $S$ the reflexive closure of $R$ is given by

$$R \cup I_S = R \cup \{(s, s) \mid s \in S\}.$$  

Since we add precisely those pairs to the relation which have to be present for it to satisfy reflexivity this is clearly the smallest relation we can form which contains $R$ as a subset and which is reflexive. Optional Exercise 31 asks you to think about how to prove this.  

When we consider a binary relation on a small finite set then checking whether the relation is reflexive, and drawing the reflexive closure, is easy. All we have to do is to make sure that every element has an arrow from itself to itself.
Example 7.25. If we go back to the relation from Example 7.21, we see that the relation is not reflexive since, with the exception of c, no element has a connection to itself.

The reflexive closure of this relation is pictured below.\(^\text{10}\)

\[\text{The new edges are drawn in red.}\]

7.3.2 Symmetry

The next important property we consider for binary relations on a set is concerned with directedness: If we can go from one element to another, can we always go back?

**Definition 53: symmetric**

A binary relation \(R\) on a set \(S\) is **symmetric** if and only if we have

\[
\text{for all } s, s' \in S \quad (s, s') \in R \text{ implies } (s', s) \in R.
\]

The first order logic formula that describes this property for a binary relation symbol \(R\) is

\[
\forall x \forall y, (R(x, y) \rightarrow R(y, x)).
\]

Relations built around the idea of equality of a property are usually symmetric, but there are plenty of relations which are not symmetric. If we have a picture of a relation we can check whether it is symmetric by checking that every connection has an arrow at each end.

**Example 7.26.** We give some examples for relations which are symmetric, and some which are not.

(a) Consider the relation between first year students in the School where two students are related if and only if they are in the same tutorial group. If student A is in the same tutorial group as student B then student B is...
in the same tutorial group as student A, and so this relation is symmetric.

(b) Look at the relation between two members of the human population where two people are related if and only if they have the same height in centimetres. Since A having the same height as B implies that B has the same height as A this relation is symmetric.

(c) The relation between two non-zero natural numbers where m is related to n if and only if m divides n is not symmetric: 1 divides 2 but 2 does not divide 1.

(d) The relation between first year students in the School where student A is related to student B if A’s student id number is below that of student B is not symmetric:11 Indeed, if the id number for student A is below that of student B then that for student B cannot be below that for student A.

(e) The relation that relates two people if and only if they are siblings is symmetric, whereas the relation that relates two people if the first is the child of the second is not.

(f) Consider the relation between two natural numbers where m is related to n if $n = 2m$. This relation is not symmetric since 2 is related to 4 but 4 is not related to 2.

(g) The relation between elements of Lists which relates two lists if and only if they have the same number of elements is symmetric.

(h) The relation which relates two objects of class List if the method equal returns true (see Example 6.13) is symmetric.

(i) The relation of semantic equivalence for propositions is symmetric.

Again we can make a relation symmetric in a unique and minimal way. If $R$ is a binary relation on a set $S$ then the symmetric closure of $R$ is given by

$$R \cup R^{op}.$$ 

This is the smallest relation on $S$ that contains $R$ and is symmetric. Optional Exercise 31 asks you to think about how to prove this.

**Example 7.27.** If we return to Example we can see that taking the union of the given relation and its opposite we get a relation where every connection has arrows pointing both ways.

---

11This example seems to be the opposite of symmetric—to make that idea precise look at the notion of anti-symmetry defined in Section 7.4.1.
Given relation $R$

\[ \begin{array}{c}
  c & \rightarrow & d \\
  b & \rightarrow & e \\
  a & \rightarrow & \end{array} \]

Its opposite $R^{op}$

\[ \begin{array}{c}
  c & \rightarrow & d \\
  b & \rightarrow & e \\
  a & \rightarrow & \end{array} \]

Their union $R \cup R^{op}$

\[ \begin{array}{c}
  c & \rightarrow & d \\
  b & \rightarrow & e \\
  a & \rightarrow & \end{array} \]

This gives us a concise way of saying what it means for a relation $R$ to be symmetric, namely

\[ R = R \cup R^{op}. \]

This can be simplified further by the observation that the non-trivial part of this equality is that

\[ R \supseteq R \cup R^{op}, \]

and since $R$ is always a superset of itself, symmetry is equivalent to demanding that

\[ R \supseteq R^{op}. \]

Another way of expressing symmetry using these ideas is to demand that

\[ R = R^{op}. \]

Above there is an argument that symmetry is equivalent to $R \supseteq R^{op}$, and by applying the $(-)^{op}$ operator on both sides this implies

\[ R^{op} \supseteq (R^{op})^{op} = R, \]

so the two must be equal.

An alternative way of describing the symmetric closure of $R$ is given by

\[ \{(s, s') \in S \times S \mid (s, s') \in R \text{ or } (s', s) \in R\}. \]

In other words, we add exactly those pairs to the relation which have to be present for the relation to become symmetric. If the relation is defined on a small finite set then we can once again look at the graph.

**Example 7.28.** Looking once again at the relation from Example 7.21 we can perform the check suggested above to see that it is not symmetric.
For example, \((b, c)\) is in the relation but \((c, b)\) is not. We may think of its symmetric closure as being constructed by adding all the arrow tips missing, and picture it\textsuperscript{12} as follows. If we think of it this way we don’t have to draw the opposite relation as in Example 7.27.

Again if we can draw the corresponding graph it is easy to see whether the relation is symmetric: We just have to check that every arrow that is not a loop has a tip at both ends.

Exercise 147. Which of the following relations is symmetric? Justify your answer.

(a) The relation where two first year students in the School are related if their last name starts with the same letter.

(b) The relation on first year students in the School where \(A\) is related to \(B\) if \(A\) can name student \(B\) when shown a picture.

(c) The following relation on the set \(\{a, b, c, d\}\):
\[
\{(a, a), (b, c), (c, b), (b, b), (c, d), (d, c), (d, d)\}
\]

(d) The relation on \(\mathbb{N} \setminus \{0\}\) where \(m\) is related to \(n\) if and only if \(m\) and \(n\) have a common divisor other than 1.

(e) The relation on \(\mathbb{N} \setminus \{0, 1\}\) where \(m\) is related to \(n\) if and only if \(m\) divides a power of \(n\).

Whenever we know a relation to be reflexive and symmetric we can picture it using an \textit{undirected graph}. This is a graph where we only record which of the elements are connected, without worrying about the direction of that connection. We do not record an element being connected with itself, we know they all are and so there’s no reason to include that information in the picture.

\textsuperscript{12}New arrow tips drawn in red.
Example 7.29. The symmetric closure of the reflexive closure of our example relation is drawn on the left, and on the right we draw the corresponding undirected graph where redundant information (for relations known to be reflexive and symmetric) has been removed.

7.3.3 Transitivity

We require one additional property of relations to define the concept we are aiming for.

Definition 54: transitive

A binary relation \( R \) on a set \( S \) is **transitive** if and only if we have that

\[
\text{for all } s, s', s'' \in S \quad (s, s') \in R \text{ and } (s', s'') \in R \implies (s, s'') \in R.
\]

The corresponding first order formula describing this property is

\[
\forall x. \forall y. \forall z. ((R(x, y) \land R(y, z)) \rightarrow R(x, z)).
\]

Again, relations based on equality of a property are often transitive, but not always.

Example 7.30. (a) Consider the relation between first year students in the School where two students are related if and only if they are in the same tutorial group. If student \( A \) is in the same tutorial group as student \( B \) and student \( B \) is in the same tutorial group as student \( C \) then student \( A \) is in the same tutorial group as student \( C \), and so this relation is transitive.

(b) Look at the relation between two members of the human population where two people are related if and only if they have the same height in centimetres. Since \( A \) having the same height as \( B \) and \( B \) having the same height as \( C \) implies that \( A \) has the same height as \( C \) this relation is transitive.

(c) The relation between first year students in the School where student \( A \) is related to student \( B \) if \( A \)'s student id number is below that of student \( B \) is transitive: Indeed, if the id number for student \( A \) is below that of student \( B \) and that for student \( B \) is below that for student \( C \) then that for student \( A \) is below that for student \( C \).

(d) The relation on the set of all humans which relates person \( A \) to
person $B$ if and only if $A$ is a child of $B$ is not transitive, but the relation that relates $A$ to $B$ if and only if $B$ is an ancestor of $A$ is. (See also Example 7.32.)

(e) Consider the relation between two natural numbers where $m$ is related to $n$ if $n = 2m$. This relation is not transitive since 2 is related to 4 and 4 is related to 8 but 2 is not related to 8.

(f) Consider the relation between human beings where person $A$ is related to person $B$ if there is a language they both speak. This relation is reflexive and symmetric but not transitive: Person $A$ may speak English and Urdu, Person $B$ may speak English and Spanish, and Person $C$ may speak Spanish. The relation is

$$\{(A, B), (B, C)\}.$$ 

So $A$ is related to $B$, and $B$ is related to $C$, but $A$ is not related to $C$.

(g) The relation between elements of Lists which relates two lists if and only if they have the same number of elements is transitive.

(h) The relation which relates two objects of class List if the method equal returns true (see Example 6.13) is transitive.

(i) The semantic equivalence relation between propositions is transitive.

We may define the **transitive closure** of a relation $R$ on a set $S$ as adding all those pairs

$$(s_1, s_n) \to R$$

for which we can find a list of elements

$$s_1, s_2, \ldots, s_n \text{ in } S, \quad \text{with } n \geq 2,$$

such that

for all $1 \leq i \leq n - 1$ we have $(s_i, s_{i+1}) \in R$.

If we look at a relation on a small finite set then transitivity is more difficult to check than reflexivity and symmetry. What we need to check here is that for all $s, s'$ in $S$,

if there is a path from $s$ to $s'$ then there is an edge from $s$ to $s'$.

In the relation from Example 7.21 the transitive closure requires us to connect every pair of elements since there is a path from every element to every element. This is messy to draw, so we consider a different example here.

**Example 7.31.** Assume we have the relation

$$R = \{(a, a), (a, b), (a, c), (b, d), (d, e)\}.$$ 

The picture of this relation is as follows:

| 368 |
The transitive closure of this relation is given by

\[ R^n \]

Example 7.32. If we look at the relation 'is a parent of' (see Example 7.7) then its transitive closure gives us the 'is an ancestor' relation.

Instead of calculating the transitive closure of a relation in one go we can do it stepwise. For this we need to define the powers (with respect to composition of relations) of a binary relation. Consider he definition of the relational composite \( R \); from page 349, and note that since a binary relation goes from a set to the same set we may compose such a relation with itself. We may therefore give the following recursive definition. Let \( R \) be a binary relation on a set \( S \).

**Base case**: \( R^0 = I_S \).

**Step case**: \( R^{n+1} = R^n \circ R \).

An intuitive explanation of \( R^n \) is the following:

- \( s \) and \( s' \) in \( S \) are related by \( R^n \) if and only if
  - one can get from \( s \) to \( s' \) by following the relation \( R \) exactly \( n \) times.

The transitive closure of \( R \) consists of all those pairs \((s, s')\) for which it is possible to get from \( s \) to \( s' \) by following \( R \) any finite number greater than 0 times. This motivates a second recursive definition.

\(^{13}\)New edges drawn in red.
Base case \( R_n \). \( R_0 = \emptyset \).

Step case \( R_n \). \( R_{n+1} = R_n \cup R_n^{n+1} \).

\( s \) and \( s' \) in \( S \) are related by \( R_n \) if and only if one can get from \( s \) to \( s' \) by following the relation \( R \) at most \( n \) times.

Note that \( R^0 \), the identity relation on \( S \), is not involved in computing the transitive closure of a relation.

**Example 7.33.** Assume the relation \( R \) describes railway journeys, so that two stations are in the relation if and only if one can travel from the first to the second having to change trains. Then \( R^2 \) gives us pairs of stations for which it is possible to travel from the first to the second by changing exactly once, \( R^3 \) gives us pairs of stations where one has to change exactly twice, whereas \( R^3 \) gives those where one has to change at most twice.

We define

\[
R_\infty = \bigcup_{n \in \mathbb{N}} R_n = \bigcup_{n \in \mathbb{N} \setminus \{0\}} R^n.
\]

This is the relation which can be described as follows:

\( s \) and \( s' \) in \( S \) are related by \( R_\infty \) if and only if one can get from \( s \) to \( s' \) by following the relation \( R \) a finite number of times.

This gives us a way of stating that a relation \( R \) is transitive, since this is the case if and only if

\[
\forall n \in \mathbb{N} \setminus \{0\} \quad \text{we have} \quad R^n \subseteq R.
\]

**Example 7.34.** If, for example, we have the set \( \{0, 1, 2, 3, 4\} \) with the relation

\[
R = \{(0, 1), (1, 2), (2, 3), (3, 4)\}
\]

then we have the following.
\[ R^0 = \{(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)\} \]

\[ R^1 = R \]
\[ R^2 = \{(0, 2), (1, 3), (2, 4)\} \]
\[ R^3 = \{(0, 3), (1, 4)\} \]
\[ R^4 = \{(0, 4)\} \]

All \( R^n \), for \( n \geq 5 \), are empty.
We draw the relations $R_n$ for this example.

$R_0 = \emptyset$

$R_1 = R_1 \cup R_0 = R$

$R_2 = R_2 \cup R_1$

$R_3 = R_3 \cup R_2$

$R_4 = R_4 \cup R_3$

All remaining $R_n$, for $n \geq 5$, are equal to $R_5$. For this reason the last picture, showing $R_4$, shows the transitive closure of $R$. 

372
Exercise 148. Show that $R_\infty$ is the transitive closure of $R$.

Exercise 149. Which of the following relations are transitive? Justify your answers. For those which are not transitive describe the transitive closure.

(a) The relation on the set of first year students within the School which relates student $A$ to student $B$ if they have at least one course unit in common.

(b) The relation on the set of first year students within the School which relates student $A$ to student $B$ if they have the same nationality.

(c) The relation on $\mathbb{N}$ where $m$ is related to $n$ if and only if $m$ divides $n$.

(d) The relation on $\mathbb{N}$ where $m$ is related to $n$ if and only if $m + n$ is even.

(e) The relation on $\mathbb{N}$ where $m$ is related to $n$ if and only if $m$ and $n$ have a common divisor greater than 1, or if $m = n = 1$.

(f) The following relation on the set $\{a, b, c, d, e\}$:

$$\{(a, a), (a, b), (b, e), (e, c)\}$$

(g) The following relation on the set $\{a, b, c, d, e\}$:

$$\{(a, a), (a, b), (c, d), (d, e)\}$$

(h) The following relation on the set $\{a, b, c, d, e\}$:

$$\{(a, a), (b, b), (c, c), (e, e)\}$$

If we want to create the transitive closure of the symmetric closure of the reflexive closure of $R$ all we have to do to this procedure is to change what we do at the start. We have to change the relation we use

$$\tilde{R} = I_S \cup R \cup R^{op}$$

This achieves two objectives:

- By adding all elements of the identity relation we add all pairs $(s, s)$. Where $s \in S$, we make sure the relation we produce is reflexive.

- By adding all elements of the relation $R^{op}$ we add all the pairs $(s', s)$ for which $(s, s') \in R$, which ensures that we produce a symmetric relation.

Proposition 7.4
The transitive closure of the symmetric closure of the reflexive closure of a relation $R$ on a set $S$ is given by $R_\infty$. 

373
Proof. By Exercise 148 we know that given a relation $R'$ the relation $R'_\infty$ is transitive. We observe that forming $I_S \cup R$ is the reflexive closure of $R$, and that the symmetric closure of the result is

\[(I_S \cup R) \cup (I_S \cup R)^{op} = (I_S \cup R) \cup (I_S^{op} \cup R^{op}) = I_S \cup R \cup R^{op} \quad I_S^{op} = I_S.\]

The remainder follows from the following exercise.

Note that the procedure of forming the reflexive symmetric transitive closure is important. If somebody tells you about particular instances they want you to consider as equivalent then this allows you to generate an equivalence relation which makes all the specified entities equivalent, but does not identify anything unnecessarily. This procedure is used in Section 7.3.4.

Exercise 150. The various closures of relations defined above work well together.

(a) Show that the symmetric closure of a reflexive relation is reflexive.

(b) Show that the transitive closure of a reflexive relation is reflexive.

(c) Show that the transitive closure of a symmetric relation is symmetric.

Conclude that the transitive closure of the symmetric closure of a reflexive relation is reflexive, symmetric and transitive. It is usually referred to as the reflexive symmetric transitive closure. Hint: You may want to use Exercise 148, but you don’t have to.

This exercise shows that if we form the transitive closure of the symmetric closure of the reflexive closure of a relation then the result will be reflexive, symmetric, and transitive.

Optional Exercise 31. Above we talk about how to add a minimal number of elements to a relation to make it reflexive, symmetric or transitive. In this exercise we make these ideas precise.

(a) Show that if $R'$ is a relation on a set $S$ which is reflexive and which contains the relation $R$ as a subset then $R'$ contains the reflexive closure of $R$.

(b) Show that if $R'$ is a relation on a set $S$ which is symmetric and which contains the relation $R$ as a subset then $R'$ contains the symmetric closure of $R$.

(c) Show that if $R'$ is a relation on a set $S$ which is transitive and which contains the relation $R$ as a subset then $R'$ contains the transitive closure of $R$.

(d) Conclude that the reflexive/symmetric/transitive closure of a re-
Optional Exercise 32. There is another way of defining the three closure operations for relations. Instead of adding elements to the given relation one may think about starting with a large relation and then removing all those pairs of elements which are not required.

(a) Show that the intersection of arbitrarily many reflexive relations is reflexive.

(b) Show that the intersection of arbitrarily many symmetric relations is symmetric.

(c) Show that the intersection of arbitrarily many transitive relations is transitive.

(d) Prove that the intersection of all reflexive/symmetric/transitive relations containing a relation \( R \) is the smallest reflexive/symmetric/transitive relation containing \( R \) and conclude (with the help of the previous optional exercise) that this intersection is equal to the reflexive/symmetric/transitive closure of \( R \).

7.3.4 Equivalence relations defined

**Definition 55: equivalence relation**

A binary relation on a set is an equivalence relation if it is reflexive, symmetric and transitive.

A number of examples are given above, but we put them together here:

**Example 7.35.** We give examples of equivalence relations.

(a) From every function we get an equivalence relation by relating those elements of the source set which are mapped to the same element in the target set, see Exercise 152.

A number of the relations mentioned above fall under this idea.

(i) Considering building blocks of the same shape equivalent has the underlying function which maps a block to its shape.

(ii) Mapping people to their height in centimetres leads to identifying those that have the same height.

(iii) Mapping students to their tutorial group allows us to identify the members of the same group.

(iv) Mapping people to their nationality\(^{14}\) allows us to talk about the nationalities represented in a particular group.

(v) Another example taken from programming is the following: Given a specific algorithm there are many programs which implement that algorithm in a particular programming language.\(^{15}\) Typically we
don’t care which particular program is used, only that it implements the chosen algorithm correctly. The underlying function here maps each program to the algorithm it implements.

(b) Relating elements of a set which have a particular property is usually a special case of the previous example, because we can map the elements of the set to the corresponding property (for example nationality, or tutorial group). But one has to be careful here in cases where there is no underlying function: Speaking a common language is not an equivalence relation since this need not be a transitive relation, see above on page 368.

(c) One might want to identify all the sets which have the same size, which leads to the notion of cardinal in mathematics.

(d) When we consider which algorithm to use to solve a particular problem we may be worried about its complexity only. In that case we typically don’t worry about distinguishing between algorithms that are in the same complexity class.

(e) In COMP11212 and COMP26120 you will learn about the notion of ‘big O’. There is an underlying equivalence relation on functions where \( f \) and \( g \) are equivalent if and only if

- there exists \( m \in \mathbb{N} \) such that \( nf \) eventually dominates \( g \) and
- there exists \( n \in \mathbb{N} \) such that \( ng \) eventually dominates \( f \).

This is an equivalence relation.

(f) The notion of semantic equivalence from Chapter 3 defines an equivalence relation on propositions, where we only care about the boolean interpretations of a given proposition.

(g) The relation between elements of \( \text{Lists}_S \) which relates two lists if and only if they have the same number of elements is another example.

(h) The relation which relates two objects of class \( \text{List} \) if the method \( \text{equal} \) returns \( \text{true} \) (see Example 6.13) is an equivalence relation.

CExercise 151. Which of the following relations are equivalence relations? Justify your answers.

(a) The relation where two first year students in the School are related if their last name starts with the same letter.

(b) Two first year students in the School being related if there is a university society they both belong to.

\(^{14}\)This ignores the possibility of dual citizenship.

\(^{15}\)Formally defining what this means is tricky, but you should get the general idea.

\(^{16}\)Formally one has to be a bit careful since there is no such thing as the ‘set of all sets’, but the idea remains.

\(^{17}\)A formal definition of what that means is given in COMP11212.
(c) The \( \equiv \) relation on propositional formulae from Chapter 4.

(d) The relation on Java objects of the class `java.lang.Object` defined by object A being related to object B if and only if the default instance method `A.equals(B)` returns true.

(e) The relation on objects of the class `BTree`, where \( t_1 \) is related to \( t_2 \) if and only if the following method (compare Code Example 6.7, when called as `equal(t1,t2)`), returns true:

```
public static boolean equal(BTree t1, BTree t2)
{
    if (t1 == null)
        return (t2 == null);
    else {
        if (t2 == null)
            return false;
        else
            return ((t1.value == t2.value)
                    && equal(t1.left, t2.left)
                    && equal(t1.right, t2.right));
    }
}
```

(f) The following relation on the set \{a, b, c, d, e\}:
\[
\{(a,a), (b,b), (a,b), (b,a), (a,c), (c,a), (b,c), (c,b), (d,d), (e,e)\}
\]

(g) The reflexive closure of the following relation on the set \{a, b, c, d, e\}:
\[
\{(a,b), (b,a), (b,c), (c,b)\}
\]

(h) The reflexive closure of the following relation on the set \{a, b, c, d, e\}:
\[
\{(a,b), (b,a)\}
\]

(i) The relation on \( \mathbb{N} \) where \( m \) is related to \( n \) if and only if \( m + n \) is even.

(j) The relation on \( \mathbb{N} \) where \( m \) is related to \( n \) if and only if \( m \) and \( n \) have a common divisor greater than 1 or if \( m = n = 1 \).

(k) The relation on \( \mathbb{N} \setminus \{0\} \) where \( m \) is related to \( n \) if and only if
\[
m \mod n = 0.
\]

**Exercise 152.** We look at the idea that functions generate equivalence relations. Assume that \( f : S \rightarrow T \) is a function.
(a) Show that the following defines an equivalence relation on \( S \):

\[ s \sim_f s' \quad \text{if and only if} \quad fs = fs'. \]

(b) Show that \( f \) is injective if and only if the corresponding equivalence relation \( \sim_f \) from (a) is given by the identity relation on \( S \),

\[ I_S = \{(s, s) \mid s \in S\}. \]

(c) Show that if \( g : T \to U \) is a function then

\[ s \sim_{g \circ f} s' \quad \text{if and only if} \quad s \sim_f s' \text{ or } fs \sim_g fs'. \]

Given a binary relation \( R \) on a set the **equivalence relation generated by a binary relation** \( R \) is the relation obtained by forming the transitive closure of the symmetric closure of the reflexive closure of \( R \). The resulting relation is reflexive, symmetric and transitive by Exercise 150. On page 373 there is a description of how to calculate that closure step-by-step. See Example 7.39 for a concrete example of carrying out the procedure.

We use an equivalence relation wherever we wish not to distinguish between certain elements of a set.

**Example 7.36.** When we use fractions to refer to rational numbers we do this with respect to the following equivalence relation: For \( m, n, m', n' \) in \( \mathbb{Z} \) we set

\[ \frac{m}{n} \sim \frac{m'}{n'} \quad \text{if and only if} \quad mn' = m'n. \]

One typically writes

\[ \frac{m}{n} = \frac{m'}{n'} \]

in that situation. How to define the rational numbers formally, and how that connects with this idea of fractions, is explained in Section 7.3.6.

### 7.3.5 Equivalence classes

Often when we have an equivalence relation on a set we do not wish to distinguish between those elements of the set which are equivalent. What we really want to do is to construct another set where there is only one representative for each class of equivalent elements.

Given a set \( S \) with an equivalence relation \( R \) and an element \( s \) of \( S \) the **equivalence class with respect to** \( R \) **generated by** \( s \), \([s]\), is the subset of \( S \) consisting of all elements of \( S \) which are related to \( s \) by \( R \), that is

\[ [s] = \{s' \in S \mid (s, s') \in R\}. \]

Note that since an equivalence relation is symmetric this is the same as

\[ \{s' \in S \mid (s', s) \in R\}. \]

If the equivalence relation in question is not clear from the context, the equivalence class generated by \( s \) for a relation \( R \) is written as \([s]_R\). Usually equivalence relations are written in infix notation and for the remainder of this section that is the notation we use.
Example 7.37. We go through the examples given in Example 7.35.

(a) For the relations given by functions we have the following equivalence classes:

(i) building blocks of the same shape,
(ii) groups of people of the same height (up to the nearest centimetre),
(iii) tutorial groups,
(iv) all people of the same nationality,
(v) implementations of the same algorithm.

(b) This example is too general to give the equivalence classes.

(c) All sets which have the same size as a given form an equivalence class (for example, all sets with 5 elements, or all countably infinite sets).

(d) Algorithms which belong to the same complexity class form an equivalence relation.

(e) All propositions with the same boolean interpretation (with respect to every valuation) form an equivalence class.

(f) Functions are in the same equivalence class if they eventually dominate each other, up to a factor.

(g) Objects of class List are in the same equivalence class for equal if and only if the underlying elements of ListsZ have the same elements in the same order.

Alternatively we can describe equivalence classes for this relation as follows. Two objects l1 and l2 of the class List are in the same equivalence class if and only if

• their instance variables l1.value and l2.value give the same integer value and
• their instance variables l1.next and l2.next are references to lists which are in the relation.

Example 7.38. The last example deserves another look, in particular if we look at the second description. This suggests that there is a recursive procedure for deciding whether two objects are in this relation, and indeed there is. We use the idea of defining a set recursively, see Section 6.4.3.

Base case $E_{\text{List}}(\text{null}, \text{null}) \in E_{\text{List}}$.

Step case $E_{\text{List}}((n, l_{1}), (m, l_{2})) \in E_{\text{List}}$ and $m == n$ for $m: \text{int}$ and $n: \text{int}$ implies $\ (\text{new List (m,l_{1}), new List (n,l_{2})}) \in E_{\text{List}}$. 

379
The base case tells us that two `List` objects that are a `null` reference are considered equivalent, and using the step case repeatedly we can build up to longer and longer lists being considered equal.

Equivalence classes split the given set into disjoint blocks of equivalent elements and we say that they *partition* the set. In other words, they give us a new set where we no longer distinguish between equivalent elements. If we take the equivalence classes of a set $S$ with respect to an equivalence relation $\sim$ the result is often written as $S/\sim$. Sometimes $S/\sim$ is referred to as the *quotient of $S$ with respect to $\sim$*.

**Example 7.39.** We begin our study of formal examples with a relation on a small finite set \{a, b, c, d, e, f\}, namely

$$\{ (a, a), (a, b), (a, e), (c, d), (d, c) \}.$$ 

As before we picture the relation using a directed graph.

```
    f
   /
  b /  e /  d
 /    /    /
 a +---+---+---+---+---+
 |   |   |   |   |   |
  c   c   c   c   c   c
```

The reflexive closure of this relation adds connections to itself for each element:

```
    f
   /
  b /  e /  d
 /    /    /
 a +---+---+---+---+---+
 |   |   |   |   |   |
  c   c   c   c   c   c
```

The symmetric closure of the relation turns all edges between different elements into double-tipped ones:

```
    f
   /
  b /  e /  d
 /    /    /
 a +---+---+---+---+---+
 |   |   |   |   |   |
  c   c   c   c   c   c
```
If we draw the result as an undirected graph we lose some of the now redundant information:

```
  f
 / \
 b   e
 |   |
 a   d
 |
 c
```

The transitive closure of this relation, both as a directed and as an undirected graph:

```
  f
 / \
 b   e
 |   |
 d   c
 |
 a
```

We draw the equivalence classes in both graphs:

```
 [f] = [a] = [b] = [e] = \{a, b, e\},  [c] = [d] = \{c, d\},  [f] = \{f\}.
```

Note the following: If we pick any pair of nodes in an equivalence class then there is a connection between them (and this is a two-sided connection in the case of a directed graph). This is true for all equivalence classes in all equivalence relations.

**Example 7.40.** If you are writing a programme where queues are implemented, and there is one central resource that all existing queues need to access (for example processing time) then you might want to implement a procedure which allocates the resource to the longest queue. From the point of view of that program, it is only important how long the queues
are, and not what elements they have. We use this idea, but for our previously defined type of list.

We define a binary relation on \( \text{Lists}_S \) where

\[
l \sim l' \quad \text{if and only if} \quad \text{len} \ l = \text{len} \ l',
\]

using the \( \text{len} \) function from Exercise 110.

Two elements of \( \text{Lists}_S \) are in the relation \( \sim \) if and only if they have the same number of elements.

One can define a corresponding relation on objects of class \( \text{List} \) where

\[
l_1 \quad \text{is related to} \quad l_2
\]

if and only if the following method, called as

\[\text{length}(l_1) == \text{length}(l_2),\]

where \( \text{length} \) is the method from Section 6.1.3 defined in detail in Exercise 110.

Note that whenever a set is partitioned, that is, split into disjoint subsets, there is an equivalence relation at the heart: Given a set \( S \), all we have to do is to define

\[
s \sim s' \quad \text{if and only if} \quad s \text{ and } s' \text{ are in the same partition.}
\]

In this case the equivalence classes are exactly the partitions. This means that partitioning a set is exactly the same thing as forming the equivalence classes for an equivalence relation.

**Exercise 153.** For the following relations, calculate the equivalence relation they generate, try to describe the resulting equivalence classes, and count their number.

(a) The reflexive symmetric transitive closure of the following relation on the set \( \{a, b, c, d, e, f\} \):

\[
\{(a, b), (b, a), (b, c), (d, e), (e, f)\}.
\]

(b) The reflexive symmetric transitive closure of the following relation on the same set as in the previous part:

\[
\{(a, a), (b, c)\}.
\]

(c) The reflexive symmetric transitive closure of the following relation on the same set as in the previous part:

\[
\{(a, a), (b, c), (c, d), (d, e)\}.
\]

(d) On the set \( \mathbb{N} \) the relation \( m \sim n \) if and only if \( m \mod 4 = n \mod 4 \). What is \([1]\) in this example? Can you describe \( \mathbb{N}/\sim \)?

382
(e) On the set $D$ of decimal numbers (that is, numbers considering of a finite number of digits from 0–9 followed by a decimal point, followed by a possibly infinite number of digits 0–9) the reflexive symmetric closure $\sim$ of the relation where two numbers are related if and only if:

- the first number ends with infinitely many 0s,
- the second number ends with infinitely many 9s and
- the (finitely many) digits to the left of these are equal, with the exception of the right-most such digit, which is one less for the second number.

What is $[1]$ in this example? Can you describe $D/\sim$?

(f) On the natural numbers $\mathbb{N}$ the relation where $m \sim n$ if $m + n$ is even (compare Exercise 151). Can you describe $\mathbb{N}/\sim$?

(g) On the complex numbers the relation where $a + bi$ is related to $a' + b'i$ if and only if $a = a'$.

(h) On the complex numbers the relation where $z$ is related to $z'$ if and only if $zz' = z'z$.

---

**Example 7.41.** We recursively define a binary relation on the set $\text{Lists}_S$ of lists over the set $S$ as follows.

**Base case** $\sim$. $[] \sim []$

**Step case** $\sim$. For $l \sim l'$ and $s, s' \in S$ we have

$s : l \sim s' : l'$.

The remainder of this example is concerned with understanding what this relation does. We show first of all that it is a reflexive relation, that is, that each list is related to itself. This is a proof by induction.

**Base case** $\sim$. We note that

$[] \sim []$  

by base case $[]$.

**Ind hyp.** We assume the statement holds for the list $l$, that is $l \sim l$.

**Step case** $\sim$. We see that given $s \in S$ we have that

$s : l \sim s : l'$  

by step case $\sim$.

We can also show by induction that this relation is also symmetric and transitive. We add a proof for the former.

\[\text{(For the finite examples you should list all equivalence classes.)}\]
**Base case** \(\sim\). We note that

\[
\text{[]} \sim \text{[]}
\]

base case \[],

and so the only instance of using the base case of the definition of \(\sim\) results in a symmetric relation.

**Ind hyp.** We assume the statement holds for the lists \(l\) and \(l'\), that is \(l \sim l'\) implies that \(l' \sim l\).

**Step case** \(\sim\). We see that the only way of building further instances of the relation \(\sim\) is to take note that, for \(l \sim l'\), as well as \(s\) and \(s'\) in \(S\) we get

\[
s : l \sim s' : l'\quad\text{by step case } \sim,
\]

but we also get, from the induction hypothesis, that \(l' \sim l\), and so

\[
s' : l' \sim s : l\quad\text{by step case } \sim,
\]

and so the relation remains symmetric as we add additional instances.

So what does this relation do? One possibility is to look at some examples, and let’s assume that the underlying set \(S\) is \(\mathbb{N}\). We know that we start with

\[
\text{[]} \sim \text{[]}.
\]

The step case tells us that we can now add any one element to \(\emptyset\) and all the resulting lists are related, so

\[
\text{[0]} \sim \text{[1]} \sim \text{[2]}, \ldots
\]

and all lists of length one are related. We can take any two of these, and add an element to each, and get two more related lists, which means that all lists of length 2 are related.

This gives rise to the conjecture that for all lists \(l\) and \(l'\) we have

\[
l \sim l'\quad\text{if and only if}\quad\text{len}\ l = \text{len}\ l'.
\]

We give a formal proof of this. This is an ‘if and only if’ statement and we show it by proving each part separately.

We first show that if \(l \sim l'\) then \(l\) and \(l'\) have the same length.

**Base case** \(\sim\). We have the base case of the relation,

\[
\text{[]} \sim \text{[]},
\]

and we can see that both sides are equal, so applying the length function gives the same result.
**Ind hyp.** For the lists \( l \) and \( l' \) we have that \( l \sim l' \) implies \( \text{len}(l) = \text{len}(l') \).

**Step case \( \sim \).** If we have lists \( l \) and \( l' \) with \( l \sim l' \) and for \( s \) and \( s' \) in \( S \) we use the step case of \( \sim \) to derive that

\[
s : l \sim s' : l'
\]

we may conclude that

\[
\text{len}(s : l) = 1 + \text{len}(l) \quad \text{step case len}
\]

\[
= 1 + \text{len}(l') \quad \text{ind hyp}
\]

\[
= \text{len}(s' : l') \quad \text{step case len}
\]

In the other direction we want to show that if for two lists \( l \) and \( l' \) we know that \( \text{len}(l) = \text{len}(l') \) then we have \( l \sim l' \). How do we know that the two lengths are equal? This can only happen by another inductive process: The length of a list can only be 0 if the list is empty, which gives the base case. If we have two lists \( l \) and \( l' \) of equal lengths, then for \( s \) and \( s' \) in \( S \) we have

\[
\text{len}(s : l) = 1 + \text{len}(l) \quad \text{step case len}
\]

\[
= 1 + \text{len}(l') \quad \text{assumption}
\]

\[
= \text{len}(s' : l').
\]

**Base case \( \sim \).** We have the base case of equal length

\[
\text{len}([]) = \text{len}([]),
\]

and we can see that

\[
[] \sim []
\]

by the base case of \( \sim \).

**Step case.** If we have lists \( l \) and \( l' \) of equal length, which allows us, for \( s \) and \( s' \) in \( S \), to conclude that

\[
\text{len}(s : l) = \text{len}(s' : l'),
\]

then we get that \( l \sim l' \) from the induction hypothesis and we may conclude that

\[
s : l \sim s' : l'
\]

by the step case of \( \sim \).

Understanding recursive definitions is not easy, and typically one wants to find an alternative description that is easier to grasp.

**CExercise 154.** Consider the following relation on \( \text{FBTrees}_S \).
Base case $\sim$. For all $s, s' \in S$ we have

$$\text{tree } s \sim \text{tree } s'.$$

Step case $\sim$. For $t \sim t', t'' \sim t'''$ and $s, s' \in S$ we have

$$\text{tree}_s(t, t'') \sim \text{tree}_{s'}(t', t''').$$

(a) Which of the following trees are $\sim$-related?

(i) $\text{tree}_2(\text{tree}_1, \text{tree}_3)$

(ii) $\text{tree}_3(\text{tree}_2, \text{tree}_3(\text{tree}_2, \text{tree}_1))$

(iii) $\text{tree}_3(\text{tree}_2(\text{tree}_1, \text{tree}_1), \text{tree}_2(\text{tree}_3, \text{tree}_2))$.

(iv) $\text{tree}_4(\text{tree}_2(\text{tree}_1, \text{tree}_3), \text{tree}_2(\text{tree}_3, \text{tree}_1))$.

(v) $\text{tree}_5(\text{tree}_2(\text{tree}_1, \text{tree}_3), \text{tree}_1)$.

(vi) $\text{tree}_3(\text{tree}_2(\text{tree}_1, \text{tree}_1), \text{tree}_2(\text{tree}_3, \text{tree}_2))$.

(vii) $\text{tree}_3(\text{tree}_2, \text{tree}_3)$

*Hint: You may want to draw them and then think about which trees are related. Begin with trees related by the base case, and then apply the step case once or twice to see how it all works.*

(b) Prove by induction that the relation is reflexive. In fact, it is an equivalence relation and you may use this to answer the following part.

(c) Informally describe the equivalence classes of this relation. How would you describe all the trees that are equivalent to a given one?

(d) How would you implement a corresponding relation for objects of class `BTree`? You are trying to write a method with the following first line:

```
public static boolean similar(BTree t1, BTree t2)
```

The method should return `true` exactly when the two input trees are related by $\sim$.

**Exercise 155.** Show the following for an arbitrary equivalence relation $\sim$ on a set $S$.

(a) For all $s \in S$ we have $s \in [s]$.

(b) For all $s, s', s''$ in $S$ we have

$$s', s'' \in [s] \implies s' \sim s''.$$ 

(c) For all $s$ and $s'$ in $S$ we have that

$$s \sim s' \implies [s] = [s'].$$

386
(d) For all $s$ and $s'$ in $S$ we have

$$[s] \cap [s']$$

is either empty or equal to $[s]$.

(e) For every element of $S$ there is an equivalence class it belongs to.

(f) The equivalence classes for the relation split $S$ into a pairwise disjoint collection of sets.

Hence the equivalence classes of $S$ partition the whole set into disjoint subsets.

While it may be useful to determine a new set which has only one representative for each equivalence class, this construction becomes significantly more useful when we have sets with operations on them. The following sub-section cover relevant examples.

### 7.3.6 Important examples

#### Modular arithmetic

When we calculate ‘modulo’ a given number we are not really interested in the numbers involved, just in the remainder they leave when dividing by the given number.

**Example 7.42.** The simplest example is calculating modulo 2, that is, we are only concerned whether the number in question is odd or even. For $i$ and $j$ in $\mathbb{N}$ we set

$$i \sim j \quad \text{if and only if} \quad i \mod 2 = j \mod 2.$$ 

It is easy to check that this is an equivalence relation. We have two equivalence classes for this relation,

- the even numbers, all of which leave remainder 0 when divided by 2 and which all are in $[0]$,
- the odd numbers, all of which leave remainder 1 when divided by 2 and all of which are in $[1]$.

So we have a new set

$$\mathbb{N}/\sim = \{[0], [1]\}.$$ 

We can calculate with these equivalence classes by adding or multiplying them, defining

$$
\begin{array}{c|cc}
+ & [0] & [1] \\
\hline
[0] & [0] & [1] \\
\end{array}
\quad
\begin{array}{c|cc}
\cdot & [0] & [1] \\
\hline
[0] & [0] & [0] \\
\end{array}
$$

We may justify this by observing that the sum of two numbers is even if and only if

- they’re both even,
• or they’re both odd,

and that the product of two numbers is odd if and only if

• they are both odd.

But, in fact, these operations may be derived from the addition and multiplication operations that exist on the set we started with, \( \mathbb{N} \).

Note that if we have two pairs of numbers, say \( i, j, i' \) and \( j' \) in \( \mathbb{N} \) with the property that

\[
i \sim i' \quad \text{and} \quad j \sim j'\]

then both,

\[
i + j \sim i' + j' \quad \text{and} \quad ij \sim i'j'.\]

To show this we have to go through all the possible cases:

• If \( i \) and \( j \) are both even then so are \( i' \) and \( j' \), and the sum of \( i \) and \( j \) is even, as is that of \( i' \) and \( j' \). In that case the product of \( i \) and \( j \) is also even, as is that of \( i' \) and \( j' \).

• If \( i \) is even and \( j \) is odd, then \( i' \) is also even, and \( j' \) is also odd. The sum of \( i \) and \( j \) is odd, as is that of \( i' \) and \( j' \), and the product of \( i \) and \( j \) is odd, as is that of \( i' \) and \( j' \).

• If \( i \) is odd and \( j \) is even then by commutativity of addition and multiplication on \( \mathbb{N} \) the argument from the previous case applies.

• If \( i \) and \( j \) are both odd then so are \( i' \) and \( j' \), and the sum of \( i \) and \( j \) is even, as is that of \( i' \) and \( j' \). The product of \( i \) and \( j \) is odd as is that of \( i' \) and \( j' \).

What this means is that in order to find the result of

\[
[i] + [j]
\]

all we have to do is pick any element of \([i]\), say \(i'\), and any element of \([j]\), say \(j'\), and calculate \(i' + j'\)—the result \([i] + [j]\) we are looking for is \([i' + j']\).

In other words, if we define

\[
[i] + [j] = [i + j],
\]

then this definition works. This is a non-trivial observation for the following reason:
What we have proved above is that, for example, we can pick any number in $[0]$, say 6, and any number in $[1]$, say 17, we can add them to each other and the result will tell us the result of $[0] + [1]$:

$$[0] + [1] = [6] + [17] = [23] = [1].$$

We might say that our equivalence relation is well-behaved for our given operations of addition and multiplication, and mathematicians might say that it is a congruence relation with respect to both $+$ and $\cdot$.

**Exercise 156.** Determine the properties of both, multiplication and addition, for $\mathbb{N}/\sim$. Are they associative or commutative? Do they have a unit? Do inverses exist for them? How does this compare with the properties of the corresponding operations for $\mathbb{N}$?

Mathematicians have a name for the set of equivalence classes from Example 7.42 with these operations, they call it the two-element field $\mathbb{F}_2$. Calculating modulo 2, that is, with odd and even numbers, is just one of infinitely many such cases.

**Example 7.43.** We may look at calculating modulo 3, for $i$ and $j$ in $\mathbb{N}$ setting

$$i \sim j \quad \text{if and only if} \quad i \mod 3 = j \mod 3.$$ 

Like many relations based on an equality this is an equivalence relation as well. There are now three equivalence classes, namely

- the numbers divisible by 3, all of which are in $[0]$,
- the numbers which leave a remainder of 1 when divided by 3, all of which are in $[1]$ and
- the numbers which leave a remainder of 2 when divided by 3, all of which are in $[2]$.

Again we would like to add and multiply these equivalence classes, but it is a bit tedious to show separately for each $n$ that calculating modulo $n$ is safe, so we do it once and for all.

**Proposition 7.5**

Let $n$, $k$, $l$, $k'$ and $l'$ be natural numbers with the property that

$$k \mod n = k' \mod n \quad \text{and} \quad l \mod n = l' \mod n.$$

Then

$$(k + l) \mod n = (k' + l') \mod n$$

and

$$kl \mod n = k'l' \mod n.$$
Proof. We know from Fact 2 that there exist unique \( m, m', i, i', j \) and \( j' \) in \( \mathbb{N} \) with
\[
0 \leq m, m' \leq n - 1
\]
and
\[
k = in + m \quad k' = i'n + m \quad l = jn + m' \quad l' = j'n + m'.
\]
We calculate
\[
k + l = in + m + jn + m' = (i + j)n + m + m'
\]
and conclude\(^{19}\) that
\[
(k + l) \mod n = (m + m') \mod n.
\]
We further calculate
\[
k' + l' = i'n + m + j'n + m' = (i' + j')n + m + m'
\]
and conclude that
\[
(k' + l') \mod n = (m + m') \mod n = (k + l) \mod n.
\]
We next calculate
\[
kl = (in + m)(jn + m')
\]
\[
= ijn^2 + im'n + jmn + mm'
\]
and conclude that
\[
kl \mod n = mm' \mod n.
\]
Similarly we calculate
\[
k'l' = (i'n + m)(j'n + m')
\]
\[
= i'j'n^2 + i'm'n + j'mn + mm'
\]
and so
\[
k'l' \mod n = mm' \mod n = kl \mod n
\]
as required.

Hence for every number \( n \) we may calculate with equivalence classes modulo \( n \). Moreover, the resulting operations are commutative and associative, and have a unit. To see commutativity, note that for all \( i \) and \( j \) in \( \mathbb{N} \) we have
\[
[i] + [j] = [i + j] = [j + i] = [j] + [i].
\]
\(^{19}\)This uses that for all \( k, i \) and \( n \) in \( \mathbb{N} \) we have \( k \mod n = (in+k) \mod n \), see Exercise 25.
The argument for associativity is similar. The unit for addition is \([0]\) since for all \(i\) in \(\mathbb{N}\) we have
\[
[i] + [0] = [i + 0] = [i] = [0 + i] = [0] + [i].
\]
The unit for multiplication is \([1]\), and the proof is very similar.

**Example 7.44.** We continue Example 7.43 by giving the tables for calculating modulo 3:

<table>
<thead>
<tr>
<th>+</th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
</tr>
<tr>
<td>[1]</td>
<td>[1]</td>
<td>[2]</td>
<td>[0]</td>
</tr>
<tr>
<td>[2]</td>
<td>[2]</td>
<td>[0]</td>
<td>[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\cdot</th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[0]</td>
<td>[0]</td>
<td>[0]</td>
</tr>
<tr>
<td>[1]</td>
<td>[1]</td>
<td>[0]</td>
<td>[1]</td>
</tr>
<tr>
<td>[2]</td>
<td>[2]</td>
<td>[0]</td>
<td>[2]</td>
</tr>
</tbody>
</table>

**Exercise 157.** Carry out the following studies.

(a) Give tables for the addition and multiplication of equivalence classes when calculating modulo 4. Identify units for addition and multiplication if they exist. Determine whether you have inverses for addition and multiplication.

(b) Give tables for the addition and multiplication of equivalence classes when calculating modulo 5. Identify units for addition and multiplication if they exist. Determine whether you have inverses for addition and multiplication.

(c) Give tables for the addition and multiplication of equivalence classes when calculating modulo 6. Identify units for addition and multiplication if they exist. Determine whether you have inverses for addition and multiplication.

*Hint:* You may want to go back to Section 2.5 to remind yourself of the notion of unit and inverse element for a binary operation.

Modular arithmetic is important in cryptography. It also appears as an important example in COMP26120. For this reason we look at more of the properties it has.

From now on we use
\[
k \sim n l \quad \text{to mean} \quad k \mod n = l \mod n.
\]

**Proposition 7.6**

The following hold for calculating modulo \(n\).

(i) For all \(n \in \mathbb{N} \setminus \{0, 1\}\), addition is commutative and associative for \(\mathbb{N}/\sim_n\); the additive unit is given by \([0]\) and every element of the set \(\mathbb{N}/\sim_n\) has an additive inverse.

(ii) For all \(n \in \mathbb{N}\setminus\{0, 1\}\), multiplication is commutative and associative for \(\mathbb{N}/\sim_n\); the multiplicative unit is given by \([1]\), and the element \(m\) of \(\mathbb{N}/\sim_n\) has a multiplicative inverse if and only if the greatest
common divisor of \( m \) and \( n \) is 1.

In particular, if \( n \) is a prime number then every number has both, an additive and a multiplicative inverse when calculating modulo \( n \).

**Proof.** We show the two parts.

(i) Associativity and commutativity of addition follow immediately from Proposition 7.5 since, for example,

\[
[k] + [l] = [k + l] \quad \text{def } + \text{ on } \mathbb{N}/\sim_n, \\
= [l + k] \quad \text{addition commutative on } \mathbb{N} \\
= [l] + [k] \quad \text{def of } + \text{ on } \mathbb{N}/\sim_n.
\]

The fact that \([0]\) is the additive unit also follows immediately from Proposition 7.5. Let \( k \) be an arbitrary element of \( \mathbb{N} \). By Fact 2 we know that \( k \mod n < n \). Hence we can set \( l = n - k \in \mathbb{N} \). Using the same proposition we calculate that

\[
[k] + [l] = [k \mod n] + [l] = [k + l] = [n] = [0]
\]

and

\[
[l + k] = [n] = [0].
\]

(ii) Commutativity and associativity of multiplication follows in the same way as for addition. The fact that \([1]\) is the multiplicative unit also follows from the Proposition 7.5. Regarding multiplicative inverses below we give a method for calculating the multiplicative inverse by giving employing Euclid’s algorithm (see Example 6.41) and from the definition of the method it is clear that such an inverse is produced whenever the two numbers we start with have 1 as their largest common divisor.

This completes the proof.

For public key cryptography the concept of *modular exponentiation* is particularly important. If we take the powers of a natural number then we eventually reach numbers that are too large for whichever format we are using. This cannot happen in modular arithmetic, because when calculating modulo \( n \) we always get a number which is below \( n \). When performing these kinds of calculation then using the square brackets for equivalence classes becomes a bit tedious. What is typically done is to merely use the numbers

\[0, 1, 2, \ldots, n - 2, n - 1,\]

which represent their respective equivalence classes. Typical notation is then

\[2 + 3 = 5 == 0 \pmod{5} \quad \text{or} \quad 2 \cdot 4 = 8 = 3 \pmod{5},\]

where previously we would have written

\[[2] + [3] = [0] \quad \text{or} \quad [2] \cdot [4] = [3],\]
where we didn’t have a good way of specifying modulo which number we were doing our calculations.

Note that when we are carrying out calculations in modular arithmetic we may, at any point, move to a different representative of the equivalence class.

**Example 7.45.** By moving to different elements of some equivalence class we can make sure that we always calculate with the smallest numbers possible.

\[ 4 \cdot 16 + 2 \cdot 17 = 64 + 34 \equiv 1 + 1 \equiv 2 \pmod{3}. \]

Note that it is customary to ensure that the final result is the canonical representative of its equivalence class, that is, when calculating modulo \( n \), a number from 0 to \( n - 1 \).

**Multiplicative inverses.** Finding multiplicative inverses modulo some given number is possible (where they exist), and we describe how this works. Assume we have a number \( n \in \mathbb{N} \setminus \{0, 1\} \), and we are looking for the multiplicative inverse of \( m \), modulo \( n \), that is we want to determine \( m^{-1} \pmod{n} \).

This means we are asking for a number \( l \), the multiplicative inverse, with the property that

\[ m \cdot l = 1 \pmod{n}, \]

which is equivalent to there being a number \( i \) such that

\[ m \cdot l = i \cdot n + 1. \]

Assume we know that the greatest common divisor of \( m \) and \( n \) is 1. If we apply Euclid’s algorithm to \( n \) and \( m \), see Example 6.41, we can see that, for \( r_0 = n \) and \( r_1 = m \), we get equalities

\[ r_0 = k_1 \cdot r_1 + r_2 \]
\[ r_1 = k_2 \cdot r_2 + r_3 \]
\[ \ldots \]
\[ r_{n-2} = k_{n-1} r_{n-1} + 1 \]
\[ r_{n-1} = k_n \cdot 1 + 0 \]

that is at some point we have

\[ r_n = 1 \]
\[ r_{n+1} = 0, \]

since the \( r_i \) that appears before one obtains 0 is the greatest common divisor of \( r_0 = n \) and \( r_1 = m \).
We take the equalities and isolate the \( r_i \) with the highest index appearing in each we obtain the following.

\[
\begin{align*}
  r_2 &= r_0 - k_1 \cdot r_1 \\
  r_3 &= r_1 - k_2 \cdot r_2 \\
  \vdots \\
  r_n &= 1 = r_{n-2} - k_{n-1} \cdot r_{n-1}
\end{align*}
\]

We can now recursively substitute each \( r_i \) in the final equality according to the previous ones, which gives us an equality that expresses 1 in terms of \( r_0 = n \) and \( r_1 = m \).

This gives the desired solution for the original problem.

**Example 7.46.** We would like to calculate the multiplicative inverse for 5 modulo 7. Following Euclid’s algorithm we calculate as follows

\[
\begin{align*}
  7 &= 1 \cdot 5 + 2 \\
  5 &= 2 \cdot 2 + 1
\end{align*}
\]

We reorganize the two equalities as instructed above to obtain

\[
\begin{align*}
  2 &= 7 - 1 \cdot 5 \\
  1 &= 5 - 2 \cdot 2.
\end{align*}
\]

We insert the right hand side of the first equality for the right hand 2 in the second equality and get

\[
1 = 5 - 2 \cdot (7 - 1 \cdot 5) = 3 \cdot 5 - 2 \cdot 7,
\]

and so

\[
3 \cdot 5 = 2 \cdot 7 + 1.
\]

Hence the multiplicative inverse of 5 modulo 7 is 3. And indeed,

\[
3 \cdot 5 = 15 = 2 \cdot 7 + 1 = 1 \pmod{7}.
\]

**Example 7.47.** We carry out another example calculation. The multiplicative inverse of 4 modulo 11 can be found as follows.

\[
\begin{align*}
  11 &= 2 \cdot 4 + 3 \\
  4 &= 1 \cdot 3 + 1
\end{align*}
\]

Rearranging gives

\[
\begin{align*}
  3 &= 11 - 2 \cdot 4 \\
  1 &= 4 - 1 \cdot 3
\end{align*}
\]
Inserting the first into the second shows that
\[ 1 = 4 - (11 - 2 \cdot 4) = 3 \cdot 4 - 11. \]
Hence the multiplicative inverse for 4 is 3, and indeed,
\[ 4 \cdot 3 = 12 = 1 \pmod{11}. \]

Example 7.48. In cases where the multiplicative inverse does not exist the algorithm proceeds as follows. Note that Euclid’s algorithm calculates the greatest common divisor of the two given numbers, Compare Example 6.41. If the greatest common divisor of the two given numbers is a number other than 1 we cannot use this idea, and indeed in that case there is no multiplicative inverse for the chosen modular arithmetic. We wish to find a multiplicative inverse for 4 modulo 6.

\[
\begin{align*}
6 &= 1 \cdot 4 + 2 \\
4 &= 2 \cdot 2 + 0.
\end{align*}
\]
In other words we reach 0 as the remainder without having reached 1 first.

Example 7.49. We give one final example. We wish to calculate the multiplicative inverse for 11 modulo 17.

\[
\begin{align*}
17 &= 1 \cdot 11 + 6 \\
11 &= 1 \cdot 6 + 5 \\
6 &= 1 \cdot 5 + 1
\end{align*}
\]
Rearranging give

\[
\begin{align*}
6 &= 17 - 1 \cdot 11 \\
5 &= 11 - 1 \cdot 6 \\
1 &= 6 - 1 \cdot 5.
\end{align*}
\]
Inserting the first two into the final equality tells us that
\[
\begin{align*}
1 &= 6 - 1 \cdot 5 & \text{given} \\
&= 6 - (11 - 1 \cdot 6) & \text{snd equality into third} \\
&= 2 \cdot 6 - 11 & \text{simplification} \\
&= 2 \cdot (17 - 1 \cdot 11) - 11 & \text{fst equality into result so far} \\
&= 2 \cdot 17 - 3 \cdot 11 & \text{simplification.}
\end{align*}
\]
So the multiplicative inverse of 11 modulo 17
\[ -3 = 14 \pmod{17}. \]
Again we can verify that

\[14 \cdot 11 = 154 = 9 \cdot 17 + 1 = 1 \pmod{17}.

Make sure that the answers you give are indeed numbers from 0 to \(n - 1\) when calculating modulo \(n\) in this notation. If in the previous example you give \(-3\) as an answer you will lose marks since this number is not an element of the set we are considering.

Note that 0 cannot have a multiplicative inverse as is shown in the following.

**Proposition 7.7**

Let \(N\) be a set with two associative binary operations, multiplication and addition, in such a way that there is a unit 0 for addition and 1 for multiplication, with \(0 \neq 1\), and such that every element has an additive inverse. Finally we assume the following distributivity law for all \(k, m\) and \(n\) in \(N\):

\[k \cdot (m + n) = k \cdot m + k \cdot n.

Then we have for all \(n \in N\) that

\[0 \cdot n = 0 = n \cdot 0,

and in particular 0 cannot have a multiplicative unit.

**Proof.** Let \(n\) be an element of \(N\).

\[
0 \cdot n = (0 + 0) \cdot n = (0 \cdot n) + (0 \cdot n)
\]

If we add the additive inverse of \(0 \cdot n\) on both sides of this equality we get

\[0 = 0 \cdot n.

Similarly we can show \(n \cdot 0 = 0\) in this situation.

Hence in this situation 0 cannot have a multiplicative inverse unless 0 is both, the additive and multiplicative unit, so 0 = 1, and that requires that \(S = \{0\}\) and that + and \(
\cdot\) are the same operation.

**Exponentiation.** The modular operation of most interest in cryptography is that of exponentiation, and we look at the basics here.

If we look at the powers of 2 \(\pmod{5}\) we find that they are

\[
\begin{array}{c|cccccccc}
\hline
n & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
2^n \pmod{5} & 1 & 2 & 4 & 3 & 1 & 2 & 4 & 3 & 1 & 2 & 4 & 3 & 1 & 2 & \ldots \\
\hline
\end{array}
\]

Of course the numbers used in cryptography are considerably larger than 5. For RSA, for example, two large primes are selected, and then exponentiation modulo their product is performed. For this and other applications it becomes important to be able to carry out exponentiation efficiently.
We look at the question of how to calculate \(a^b \pmod{n}\).

**Code Example 7.1.** The really naive method is to calculate \(a\) to the power of \(b\) as integers, and then calculate the remainder when dividing by \(n\), that is\(^{20}\)

```java
public static int modpower (int a, int b, int n)
{
    if (b==0)
        return 1;
    else {
        int power = 1;
        for (int count=1, count <= b, count++)
            power = (power * a);
        return power % n;
    }
}
```

Using this code quickly runs out of memory\(^{21}\) despite the fact that we know that the result is a number from 0 to \(n - 1\). We look at a sample calculation of \(13^5 \pmod{197}\).

The table gives the value of power as the loop progresses.

<table>
<thead>
<tr>
<th>count</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>power</td>
<td>13</td>
<td>169</td>
<td>2197</td>
<td>28561</td>
<td>371293</td>
</tr>
</tbody>
</table>

We then take the final result and calculate

\[371293 \pmod{197} = 145.\]

**Code Example 7.2.** The slightly less naive algorithm is to just keep multiplying by the base and calculating the remainder when dividing by \(n\) at each step.

```java
public static int modpower (int a, int b, int n)
{
    if (b==0)
        return 1;
    else {
        int power = 1;
        for (int count=1, count <= b, count++)
            power = (power * a);
        return power % n;
    }
}
```

\(^{20}\)Note that one could improve efficiency by checking whether \(a\) is equal to 0 or 1, but this would not improve matters much and we are looking for a short program.

\(^{21}\)even if you pick one of the large integer classes in **Java** instead of **int**.
power = (power * a) % n;

    return power;

}
}

This works better but still is not very efficient. It performs \( b \) many multiplications and remainder operations. Again, here is a sample calculation of the same number as before.

<table>
<thead>
<tr>
<th>count</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>power</td>
<td>13</td>
<td>169</td>
<td>30</td>
<td>193</td>
<td>145</td>
</tr>
</tbody>
</table>

This program does not run out of memory as the first one inevitably does.

Note that this works because of Proposition 7.5: We want to calculate

\[ a^b \pmod{n}, \]

but we know that

\[ i \cdot j \pmod{n} \]

has the same result as \( (i \pmod{n}) \star (j \pmod{n}) \),

and so taking the remainder when dividing by \( n \) at each step of the exponentiation process is the same as taking it at the end:

\[
\begin{align*}
    a^b \pmod{n} &= (a \cdot a^{b-1}) \pmod{n} \\
    &= (a \cdot (a^{b-1} \pmod{n})) \pmod{n}
\end{align*}
\]

Prop 7.5.

Indeed, one way of paraphrasing that proposition is to say that when adding or multiplying \( \pmod{n} \) one may form the remainder when dividing by \( n \) at any time!\(^\text{22}\)

Code Example 7.3. In COMP26120 you will look at a third method for performing this calculation.

```java
public static int modpower (int a, int b, int n)
{
    if (b==0)
        return 1;
    else {
        int power = 1;
        while (b != 0) {
            if (b % 2 == 1)
                power = (power * a) % n;
            a = (a+a) % n;
            b = b/2;
        }
        return power;
    }
}
```
Again we perform a sample calculation, but this time we have to track not just of the value held in the variable `power`, but also of what is held in the variables `a` and `b`, the latter of which controls the `while` loop. If that variable is even then nothing happens in the `while` loop, which we denote in the table by an empty cell for `power`. We repeat the above calculation, so at the start we have `a = 13`, `b = 5` and `n = 197`. The method sets the variable `power` to `1`, so we use that as our first value for that variable.

<table>
<thead>
<tr>
<th>b</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>power</td>
<td>1</td>
<td>13</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>13</td>
<td>169</td>
<td>193</td>
<td>16</td>
</tr>
</tbody>
</table>

Because this is a short calculation it looks as if we have just as many steps to perform as before, since we now have to calculate changing values for `a` as well. But you can see that when `b` is large the fact that the `while` loop is carried out only \( \log b \) many times becomes more important than the fact that each time we have between one and two multiplications to carry out.

The idea behind this algorithm is quite simple: Instead of multiplying with `a` the required number of times the number of multiplications is brought down by the squaring operation. To do this one may think of `b` in base 2.

**Example 7.50.** We illustrate how this works in an example: For `b = 5`, which is `101` in binary, we have

\[
a^5 = a^4 \cdot a^1 = (a^2)^2 \cdot a^1 = a^{1 \cdot 2^2} \cdot a^{0 \cdot 2^1} \cdot a^{1 \cdot 2^0},
\]

that is, we have one count of an iterated square of `a` for each position where `b` in binary has `1`. A larger example is `b = 21`, where we have

<table>
<thead>
<tr>
<th>b</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>powers</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>((a^2)^2)^2</td>
<td>(a^2)^2</td>
<td>(a)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

and so

\[
a^{21} = a^{1 \cdot 2^4} \cdot a^{0 \cdot 2^3} \cdot a^{1 \cdot 2^2} \cdot a^{0 \cdot 2^1} \cdot a^{1 \cdot 2^0}.
\]

The algorithm ensures that what is multiplied is the appropriate selection of iterated squares.

**Example 7.51.** We carry out one more example that illustrates how the two algorithms work with numbers where it’s a bit easier to follow the calculation. We calculate

\[7^7 \pmod{11}.\]

We follow the second algorithm.

---

\(^{22}\)In COMP11212 you will learn about *loop invariants* and then you can make a rigorous step-by-step argument that the two programs compute the same number.

399
For the third algorithm at the start we have \( a = 7, b = 7, n = 11 \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 7^n \pmod{11} )</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Exercise 158. Carry out the following tasks.

(a) Find the multiplicative inverses, where they exist, for

\[
7 \pmod{11}, \quad 15 \pmod{17}, \quad 8 \pmod{12}, \quad 5 \pmod{9}.
\]

\textit{Hint: See Examples 7.46 to 7.49.}

(b) Calculate the following using the two algorithms from Code Examples 7.2 and 7.3.

\[
4^5 \pmod{13}, \quad 7^4 \pmod{9}, \quad 6^5 \pmod{11}, \quad 8^6 \pmod{16}.
\]

(c) Depending on \( b \), how many times is the loop in the algorithm from Code Example 7.2 carried out? What about the other algorithm?

(d) For which of the following can you find an exponent so that the given number is equal to 1 in the relevant modular arithmetic? Give the smallest such exponent if possible, argue why there is none otherwise.

\[
7 \pmod{11}, \quad 15 \pmod{17}, \quad 8 \pmod{12}, \quad 5 \pmod{9}.
\]

\textit{Hint: There is no algorithm that determines the required number for all cases.}

The integers

The point of this and the following account of the rationals is to show you how to formally define these numbers with their operations. The material is not examinable as such, but does provide more examples for using equivalence relations and understanding it may be helpful for answering other exam questions.

Consider the following set:

\[
\mathbb{N} \times \mathbb{N} = \{(m, n) \mid m, n \in \mathbb{N}\}.
\]

On this set we define a relation, namely

\[
(m, n) \sim (m', n') \quad \text{if and only if} \quad m + n' = m' + n.
\]

It is easy to see that this is an equivalence relation: Since for all \( m \) and \( n \) in \( \mathbb{N} \) we have that \( m + n = m + n \) it is reflexive, and symmetry is obvious from the definition. Transitivity is obtained as follows:

\[
(m, n) \sim (m', n') \quad \text{and} \quad (m', n') \sim (m'', n''),
\]
imply
\[ m + n' = m' + n \] and \[ m' + n'' = m'' + n' \]
respectively, and together these imply that
\[
\begin{align*}
m + n'' &= m' - n' + m' - m' + n'' \quad & n' - n' = 0, m' - m' = 0 \\
(m + n') - (n' + m') + (m' + n'') \quad & \text{comm, ass +, Exercise 34} \\
(m'+ n) - (n' + m') + (m'' + n') \quad & (m, n) \sim (m', n'), (m', n') \sim (m'', n'') \\
m'' + m' + n' - n' + n \quad & \text{comm, ass +, Exercise 34} \\
m'' + n \quad & m' - m' = 0, n' - n' = 0,
\end{align*}
\]
which means that we have
\[ (m, n) \sim (m'', n''). \]

We define an operation we call ‘addition’ on this set, by setting
\[ (m, n) + (m', n') = (m + m', n + n'). \]

We next show that
\[ (k, l) \sim (m, n) \text{ and } (k', l') \sim (m', n') \]
implies \[ (k, l) + (k', l') \sim (m, n) + (m', n'), \]
by calculating
\[
\begin{align*}
(k, l) + (k', l') &= (k + k', l + l') \quad \text{and} \quad (m, n) + (m', n') = (m + m', n + n') \\
&\quad \text{and} \quad k + k' + n + n' = (k + n) + (k' + n') \quad \text{comm, ass +} \\
&\quad \text{and} \quad = (m + l) + (m' + l') \quad (k, l) \sim (m, n), (k', l') \sim (m', n') \\
&\quad \text{and} \quad = m + m' + l + l' \quad \text{comm, ass +}.
\end{align*}
\]

This allows us to define an addition operation on the set of equivalence classes with respect to \( \sim \), \((\mathbb{N} \times \mathbb{N})/\sim\).

Given two arbitrary equivalence classes for our set, if we pick two respective elements, say \((k, l)\) and \((m, n)\) we may set
\[ [(k, l)] + [(m, n)] = [(k, l) + (m, n)] \]
since our previous calculation assures us that no matter which elements of each equivalence class we pick, the result of adding the two always determines the same equivalence class.

It is fairly easy to show that addition is commutative and associative for our original set, \(\mathbb{N} \times \mathbb{N}\), and the same is then true for the derived operation on \((\mathbb{N} \times \mathbb{N})/\sim\). We can also check that there is a unit, namely the equivalence class of \((0, 0)\): Given \((m, n) \in \mathbb{N} \times \mathbb{N}\) we have
\[ [(m, n)] + [(0, 0)] = [(m + 0, n + 0)] = [(m, n)] = [(0, 0)] + [(m, n)]. \]
This raises the question of whether we have inverses with respect to this operation. In other words, given \((m, n) \in \mathbb{N} \times \mathbb{N}\), can we find an element whose equivalence class we can add to \([m, n]\) such that the result is \([0, 0]\)? To find the answer to this question it is a good idea to first think about what the elements of \([0, 0]\) are. An element \((m, n)\) of our set is equivalent to \((0, 0)\) if and only if

\[m + 0 = 0 + n,\]

that is, if \(m = n\).

Given \((m, n) \in \mathbb{N} \times \mathbb{N}\) we note that

\[([m, n]) + [(n, m)] = [(m + n, n + m)] = [(0, 0)],\]

and so we do indeed have an inverse element, namely \([(n, m)]\) for \([(m, n)]\). In other words, \((\mathbb{N} + \mathbb{N})/\sim\) is a commutative group with respect to addition. Indeed, this is nothing but an alternative description for a group you are all familiar with.

As a preliminary calculation we note that for \((m, n) \in \mathbb{N} \times \mathbb{N}\) we have

\[([m, n]) = [(m - n, 0)],\]

since

\[m + 0 = m = m - n + n\]

and so

\[(m, n) \sim (m - n, 0).\]

Consider the following function from \(\mathbb{N} \times \mathbb{N}\) to \(\mathbb{Z}\):

\[(m, n) \rightarrow m - n.\]

This function is constant on equivalence classes, since if we have

\[(m, n) \sim (m', n')\]

then we have

\[m - n = m + n' - n' - n = m' + n - n' - n = m' - n'.\]

Hence we may define a function \(f\) from \((\mathbb{N} \times \mathbb{N})/\sim\) to \(\mathbb{Z}\) by setting

\[f: [(m, n)] \rightarrow m - n,\]

without worrying which element of \([(m, n)]\) we may be referring to. We further define a function \(g\) from \(\mathbb{Z}\) to \((\mathbb{N} \times \mathbb{N})/\sim\) by setting

\[g: i \rightarrow [(i, 0)].\]

We claim that this defines bijection between the two sets: Given \(i \in \mathbb{Z}\) we calculate

\[i \xrightarrow{g} [(i, 0)] \xrightarrow{f} i - 0 = i,\]

and given \((m, n) \in \mathbb{N} \times \mathbb{N}\) we have

\[([m, n]) \xrightarrow{f} m - n \xrightarrow{g} [(m - n, 0)] = [(m, n)].\]
and so the two functions are mutual inverses. But much more is true: These two functions preserve the addition operation, that is, we have, for \((k, l)\) and \((m, n)\) in \(\mathbb{N} \times \mathbb{N}\) that
\[
\begin{align*}
f[(k, l)] + f[(m, n)] &= (k - l) + (m - n) \quad \text{def } f \\
&= (k + m) - (l + n) \quad \text{ass, comm +, Ex 34} \\
&= f[(k + m, l + n)] \quad \text{def } f \\
&= f([(k, l)] + [(m, n)]) \quad \text{def addition},
\end{align*}
\]
and for \(i\) and \(j\) in \(\mathbb{Z}\) we have
\[
\begin{align*}
gi + gj &= [(i, 0)] + [(j, 0)] = [(i + j, 0)] = g(i + j).
\end{align*}
\]
This tells us that our set \((\mathbb{N} + \mathbb{N})/\sim\) really is describing the set \(\mathbb{Z}\) with its addition operation by simply giving a different name to all the elements. Indeed, the formal definition of the set of integers, \(\mathbb{Z}\), is \((\mathbb{N} \times \mathbb{N})/\sim\), where we use the integer \(m - n\) as a name for the equivalence class of \((m, n)\).

We can also define multiplication on our ‘formal integers’, \((\mathbb{N} \times \mathbb{N})/\sim\) by setting
\[
\begin{align*}
[(m, n)] \cdot [(m', n')] &= [(mm' + UN', mn' + m'n)].
\end{align*}
\]

**Exercise 159.** This exercise is concerned with checking that this definition of multiplication makes sense and fits with the usual one on \(\mathbb{Z}\).

(a) Show that for \((k, l), (k', l')\), \((m, n)\) and \((m', n')\) in \(\mathbb{N} \times \mathbb{N}\) we have that \((k, l) \sim (k', l')\) and \((m, n) \sim (m', n')\) implies
\[
(k, l) \cdot (m, n) \sim (k', l') \cdot (m', n').
\]
This means that our definition of multiplication works properly on equivalence classes with respect to \(\sim\).

(b) Show that \([(1, 0)]\) is the unit for this multiplication.

(c) Show that for \(i, j \in \mathbb{Z}\) we have \((gi) \cdot (gj) = g(ij)\).

(d) Show that for \((k, l), (m, n)\) \in \(\mathbb{N} \times \mathbb{N}\) we have
\[
f[(k, l)]f[(m, n)] = f([(k, l)] \cdot [(m, n)])).
\]
Hence our formal integers can play the role of \(\mathbb{Z}\) for all the usually considered operations, namely addition (which gives subtraction thanks to the existence of inverses) and multiplication. Mathematicians say that the two are isomorphic as rings.

**Optional Exercise 33.** In the formal definition of \(\mathbb{Z}\) we use a number of properties that hold on a very general level. This exercise is about understanding those properties better. Assume you have an equivalence relation \(\sim\) on a set \(S\).

(a) Define a function \(q\) from \(S\) to \(S/\sim\) which maps each element to its equivalence relation. This is known as the quotient map.

(b) Assume you have a function \(f\) from \(S\) to a set \(T\) with the property
that, for \( s, s' \in S \) we have \( s \sim s' \) implies \( f(s) = f(s') \). Define a function \( f_{\sim} \) from \( S/\sim \) to \( T \) with the property that \( f_{\sim} \circ q = f \). Can you argue that \( f_{\sim} \) is uniquely determined by these properties?

(c) Assume you have a binary operation \( \oplus \) on \( S \) such that for \( s, s', t, t' \in S \) we have \( s \sim t \) and \( s' \sim t' \) implies \( s \oplus s' \sim t \oplus t' \) (we may think of this as saying that the operation is well-behaved with respect to the equivalence relation)\(^{23}\). Define a binary operation \( \oplus \) on \( S \) with the property that for all \( s, s' \in S \) you have

\[
qs \oplus qs' = q(s \oplus s').
\]

Can you show that this operation is uniquely determined by these properties?

(d) Assume that we have the same situation as in the previous part. Show that if the given operation is commutative or associative, then the newly defined operation has the same property. Further show that if \( e \) is a unit for the operation \( \oplus \) then \([e]\) is a unit for the operation \( \oplus \). Lastly prove that if \( s' \) is an inverse for \( s \) with respect to \( \oplus \) then \([s']\) is an inverse for \([s]\) with respect to \( \oplus \).

(e) Check that the addition and multiplication operations for \((\mathbb{N} \times \mathbb{N})/\sim\) given in the text above are special cases of such binary operations.

**Definition 56: integers**

The commutative ring \( \mathbb{Z} \) of integers is (formally) defined as the quotient of \( \mathbb{N} \times \mathbb{N} \) with respect to the equivalence relation given above, and with addition and multiplication operations also given above.

**The rationals**

We can use a similar construction to obtain a formal definition of the rational numbers. As a basis we use the set

\[
\mathbb{Z} \times (\mathbb{Z} \setminus \{0\}) = \{(i, m) \mid i, m \in \mathbb{Z}, m \neq 0\}.
\]

We use the following equivalence relation on this set:

\[
(i, m) \sim (j, n) \quad \text{if and only if} \quad in = jm.
\]

**Exercise 160.** Show that the relation defined above is an equivalence relation.

We may then form the set \((\mathbb{Z} \times (\mathbb{Z} \setminus \{0\}))/\sim\) of \( \sim \)-equivalence classes. As before we are concerned with defining operations on this new set. This time we begin with multiplication. For \( i, j \) in \( \mathbb{Z} \) and \( m, n \) in \( \mathbb{Z} \setminus \{0\} \) we set\(^{24}\)

\[
[(i, m)] \cdot [(j, n)] = [(ij, mn)].
\]

\(^{23}\) If we think of the operation as the more fundamental item we say that the relation is a congruence relation for the given operation.

\(^{24}\) Again one has to check that this makes sense—see the exercise below.
This defines a commutative and associative operation with unit \([(1,1)]\). We have inverses for this operation: Assume we have \(i, m \in \mathbb{Z}\) with \(m \neq 0\). We claim that the inverse for \([(i, m)]\) is \([(m, i)]\) and verify this by calculating
\[
[(i, m)] \cdot [(m, i)] = [(im, mi)] = [(1, 1)],
\]
and
\[
[(m, i)] \cdot [(i, m)] = [(mi, im)] = [(1, 1)].
\]
We still owe the connection between our set of equivalence classes and the set of rational numbers \(\mathbb{Q}\). This is a bit trickier to describe cleanly than the case of the integers above because effectively the rational numbers have been treated as a quotient all along: Given a rational number we know that we may find integers \(m\) and \(n\), with \(n \neq 0\) such that our number is equal to the fraction \(m/n\). But \(m\) and \(n\) are not defined uniquely, and so we effectively use an equivalence relation on fractions, where
\[
\frac{m}{n} \approx \frac{m'}{n'} \quad \text{if and only if} \quad mn' = m'n.
\]
This suggests that the proper translation between our set \((\mathbb{Z} \times (\mathbb{Z} \setminus \{0\}))/\sim\) and \(\mathbb{Q}\) is as follows. We define a function \(f\) from \((\mathbb{Z} \times (\mathbb{Z} \setminus \{0\}))/\sim\) to \(\mathbb{Q}\) by setting
\[
[(m, n)] \longmapsto \frac{m}{n},
\]
and a function \(g\) in the opposite direction by setting
\[
\frac{m}{n} \longmapsto [(m, n)].
\]

**Exercise 161.** This exercise is concerned with checking that the claims made above are true.

(a) Show that for \((i, m), (i', m'), (j, n)\) and \((j', n')\) in \(\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})\) we have that \((i, m) \sim (i', m')\) and \((j, n) \sim (j', n')\) implies \((ij, mn) \sim (i'j', m'n')\). This means that our definition of multiplication works properly on equivalence classes with respect to \(\sim\).

(b) Show that \([(1, 1)]\) is the unit for this multiplication.

(c) Observe that for \(i, j, m, n\) from \(\mathbb{Z}\) with \(m, n \neq 0\) we have
\[
(i, m) \sim (j, n) \quad \text{if and only if} \quad \frac{i}{m} \approx \frac{j}{n}
\]
and conclude that \(\approx\) is an equivalence relation. Further note that this implies that the definition of \(g\) makes sense by mapping equivalent fractions to the same equivalence class for \(\sim\) in \((\mathbb{Z} \times (\mathbb{Z} \setminus \{0\}))\).

(d) Show that for \(i, j, m, n \in \mathbb{Z}\) with \(m, n \neq 0\) we have \(g(\frac{i}{m}) \cdot g(\frac{j}{n}) = g(\frac{i}{m} \cdot \frac{j}{n})\).
(e) Show that for \((i,m), (j,n) \in \mathbb{Z} \times (\mathbb{Z} \setminus \{0\})\) we have

\[ f[(i,m)]f[(j,n)] = f([i,m] \cdot [(j,n)]). \]

This means that our \((\mathbb{Z} \times (\mathbb{Z} \setminus \{0\}))/_\sim\) really is a way of talking about the rationals cleanly, that is, we know exactly what this set looks like, where before we did not have a clean way of referring to its elements. But so far we have only considered multiplication and still owe the addition operation. We set, for \(i, j, m\) and \(n\) in \(\mathbb{Z}\), with \(m,n \neq 0\),

\[ [(i,m)] + [(j,n)] = [(im + jm, mn)]. \]

This defines a commutative associative operation with unit \([(0,1)]\), and with inverses where the additive inverse of \([(i,m)]\) is given by \([(−i, m)]\).

**Exercise 162.** This exercise is concerned with checking that this definition of addition makes sense and fits with the usual one from \(\mathbb{Q}\).

(a) Show that for \((i,m), (i',m'), (j,n)\) and \((j',n')\) in \(\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})\) we have that \((i,m) \sim (i',m')\) and \((j,n) \sim (j',n')\) implies

\[ (i, m) + (j, n) \sim (i', m') + (j', n'). \]

This means that our definition of addition works properly on equivalence classes with respect to \(\sim\).

(b) Show that \([(0,1)]\) is the unit for this addition.

(c) Show that for \(i,j,m,n \in \mathbb{Z}\) with \(m,n \neq 0\) we have

\[ g(\frac{i}{m}) + g(\frac{j}{n}) = g(\frac{i}{m} + \frac{j}{n}). \]

(d) Show that for \((i,m), (j,n) \in \mathbb{Z} \times (\mathbb{Z} \setminus \{0\})\) we have

\[ f[(i,m)] + f[(j,n)] = f([i,m] + [(j,n)]). \]

Hence we may use \((\mathbb{Z} \times (\mathbb{Z} \setminus \{0\}))/_\sim\) as a formal definition of the rational numbers (with addition and multiplication)\(^{25}\), and that is what mathematicians do. Instead of assuming that \(m/n\) has a predefined meaning, they use \((m,n)\) to express the same number, being aware of the fact that an equivalence relation is required since there is more than one way of representing a given number. A mathematician would say that what we have defined above is a **field**.

**Optional Exercise 34.** Carry out the last part of Optional Exercise 33 for multiplication and addition on \((\mathbb{Z} \times (\mathbb{Z} \setminus \{0\}))/_\sim\) as defined in this section.

\(^{25}\) Since subtraction and division are defined via inverses for addition and multiplication, respectively, these are also included.
Definition 57: rationals

The field \( \mathbb{Q} \) of rational numbers is (formally) defined as the quotient of \( \mathbb{Z} \times (\mathbb{Z} \setminus \{0\}) \) with respect to the equivalence relation given above, and with addition and multiplication operations also given above.

Further examples

Polynomial functions (see also Section 0.3.5) often come up as describing the complexity of various algorithms or programs. These functions exist over all our sets of numbers. Assume we have fixed a set of numbers \( N \).

The formal definition of a polynomial is that of a function from \( N \) to \( N \) for which we can find \( n \in \mathbb{N} \) and \( a_0, a_1, \ldots, a_n \) in \( N \) such that the function is given by

\[
\sum_{i=0}^{n} a_i x^i = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.
\]

The \( a_i \) are the coefficients. We define the degree of the polynomial to be the largest \( n \) for which \( a_n \neq 0 \), and write

\[
\text{deg} \left( \sum_{i=0}^{n} a_i x^i \right)
\]

for this number. The corresponding coefficient \( a_n \) is called the leading coefficient of the polynomial in question.

Example 7.52. We calculate some degrees.

\[
\begin{align*}
\text{deg} \left( 3x^4 + 2x^2 - 5 \right) &= 4 \\
\text{deg} \left( 0x^5 + 3x^3 - 2x^1 \right) &= 3 \\
\text{deg} \left( 0x^5 + 0x^2 - 4 \right) &= 0.
\end{align*}
\]

The leading coefficients of these polynomials are 3, 3, and \(-4\) respectively.

When a coefficient is 0 it is customary not to write the corresponding part of the term, that is, we omit expressions of the form \( 0x^i \).

We can add such polynomial functions, and we can multiply them, in an obvious way. We first look at an example for each, and then give a general definition.

Example 7.53. We look at how to add polynomial to another. In the example below we have a polynomial of degree 3 which we want to add

\[\text{It is possible to formally define polynomials without viewing them as functions. The definition of the addition and multiplication operations for polynomials given below still work in that case.}\]
to one of degree 2, for example,

\[
\begin{align*}
  x^3 + & \quad 2x + \quad 1 \\
  + & \quad 3x^2 + \quad 5x + \quad 10 \\
  = & \quad x^3 + \quad 3x^2 + \quad 7x + \quad 11.
\end{align*}
\]

In general would like to write

\[
\sum_{i=0}^{m} a_i x^i + \sum_{i=0}^{n} b_i x^i = \sum_{i=0}^{\max(m,n)} (a_i + b_i) x^i,
\]

but it may be the case that some of the coefficients are not defined. If we go back to the concrete example above we have

\[
\begin{align*}
  a_3 &= 1 & a_2 &= 0 & a_1 &= 2 & a_0 &= 1 \quad \text{and} \\
  b_2 &= 3 & b_1 &= 5 & b_0 &= 10.
\end{align*}
\]

Here the coefficient \( b_3 \) is not defined, so the convention is to assume that such coefficients are equal to 0 to make the formal definition work. Multiplication is slightly more complicated.

**Example 7.54.** Again we start with an example.

\[
(x^3 + 2x + 1)(3x^2 + 5x + 10)
\]

\[
= (1 \cdot 3)x^5 + (1 \cdot 5)x^4 + (1 \cdot 10 + 2 \cdot 3)x^3 + (2 \cdot 5 + 1 \cdot 3)x^2
\]

\[
+ (2 \cdot 10 + 1 \cdot 5)x + 1 \cdot 10
\]

\[
= 3x^5 + 5x^4 + 16x^3 + 13x^2 + 25x + 10.
\]

In general, the definition is

\[
\left( \sum_{i=0}^{m} a_i x^i \right) \left( \sum_{i=0}^{n} b_i x^i \right) = \sum_{i=0}^{m+n} \left( \sum_{j=0}^{i} a_{i-j} b_j \right) x^i,
\]

which multiplied out looks something like

\[
\begin{align*}
  a_0 b_0 &+ (a_0 b_1 + a_1 b_0)x + (a_2 b_0 + a_1 b_1 + a_0 b_2)x^2 \\
  &+ \cdots + (a_{m-1} b_n + a_m b_{n-1})x^{m+n-1} + a_m b_n x^{m+n}.
\end{align*}
\]

These operations on polynomials are associative and commutative. The unit for addition is the constant 0 polynomial, all of whose coefficients are 0, while the unit for multiplication is the constant 1 polynomial, where \( a_0 = 1 \) and all other coefficients are 0.

When we consider the complexity of an algorithm we often are only interested in the degree of the polynomial, that is, the highest power of \( x \) that occurs in the term.

We define an equivalence relation on polynomial functions, whereby

\[
\sum_{i=0}^{m} a_i x^i \sim \sum_{i=0}^{n} b_i x^i \quad \text{if and only if} \quad \deg \left( \sum_{i=0}^{m} a_i x^i \right) = \deg \left( \sum_{i=0}^{n} b_i x^i \right).
\]
In other words we consider two polynomial functions equivalent if and only if they have the same degree, that is if the largest \( n \) for which \( x^n \) has a non-zero coefficient, is the same for both. This means that two polynomials are in the same equivalence class if and only if they define the same ‘big O’ class of functions, compare Example 7.35. You are asked to think about some aspects of this idea in Exercise 165.

Now when we want to determine the complexity of an algorithm or program, and we know this is a polynomial function, we can concentrate on determining the degree. For this we need to be able to calculate with this equivalence relation, and the following exercise invites you to work out how this works. You will study these ideas in more generality in COMP11212 and COMP26120.

**Exercise 163.** For the following polynomials, give the degrees and leading coefficients, and calculate their sum and product.

- \( 2x^5 + x^3 - 1 \) and
- \( x + 1 \).

What are the degrees and coefficients of the two resulting polynomials? How do they arise from the degrees and leading coefficients of the two polynomials you started with?

**Exercise 164.** Assume you are trying to implement polynomials with coefficients from \( \mathbb{N} \) in Java by using the List class from Section 6.1, see Code Example 6.1. You want to think of the list as giving you the coefficient of the polynomial, so that

\[
[3, 5, 2]
\]

encodes the polynomial

\[
3 + 5x + 2x^2.
\]

The empty list (or null reference) does not correspond to a polynomial via this encoding. We are going to use it as another name for the polynomial of degree 0 whose (only) coefficient is 0, that is, it encodes the same polynomial as the list [0].

Give definitions for the following operations, either by recursive definition for elements of Lists\(_Z\) or by giving a recursive Java method for the class List. You have to stick to one of these for all the parts.

(a) Addition of polynomials.

(b) Multiplication of polynomials.

This is quite complicated. You may either develop your own way of doing this, or you may work according to the following idea.

When we multiply a given polynomial, say

\[
2 + x + 3x^2
\]
with another polynomial, say \( p \), we have

\[
(2 + x + 3x^2) \cdot p = \frac{}{} ; 2 \cdot p \\
+ x \cdot p \\
+ 3x^2 \cdot p.
\]

Hence in order to define multiplication of polynomials it is sufficient to use addition provided we also define a method that takes a polynomial and multiplies it by a polynomial of the form

\[ mx^n, \]

where \( m \) and \( n \) are elements of \( \mathbb{N} \).

- For multiplying a given polynomial by a polynomial of the form \( x^n \)
you may find it convenient to have an operation that takes as input
an integer \( n \) and returns a list which consists of \( n \) many 0s.
- It is useful to have an operation that takes a polynomial and multi-
plies is by a natural number (by multiplying all the coefficients by
that natural number).
- Using the previous two parts one may define an operation that
multiplies a given polynomial by a polynomial of the form

\[ mx^n, \]

where \( m, n \in \mathbb{N} \).
- Using the previous part and the addition operation for polynomials
you should now be able to define multiplication of polynomials.

You may use previously defined operations for lists at any point.

**Exercise 165.** Let \( P_\mathbb{N} \) be the set of polynomials with coefficients
in \( \mathbb{N} \).

(a) Above there is the definition of an equivalence relation \( \sim \)
on the set of polynomials which relates two polynomials if they have the same
degree. Find a way of representing these so that you can describe the
set \( P_\mathbb{N}/\sim \). Which polynomials are in \([0]\)?

(b) Define an addition operation \( \oplus \) on \( P_\mathbb{N}/\sim \). Ensure that for polynomi-
als \( p \) and \( q \) you have \([p] \oplus [q] = [p + q]\) — in other words, the equivalence
class we obtain when adding the class of \( p \) to the class of \( q \) is the class
of \( p + q \).

(c) Define a multiplication \( \otimes \) operation on \( P_\mathbb{N}/\sim \). Ensure that for poly-
nomial functions \( p \) and \( q \) you have \([p] \otimes [q] = [pq]\).

*Hint: In Exercise 157 you are asked to define addition and multiplication
operations on equivalence classes when calculating modulo some number.
You can think of this exercise as asking you to define addition and multi-

plication operations on equivalence classes of polynomials given by their degrees.
If you are struggling with this then start by carrying out Exercise 163. To get the marks for preparing the exercise that’s a requirement if you cannot do anything else.

Finite state automata have an interesting equivalence relation on them, given by bisimulations.

**Exercise 166.** Argue that the following relation is an equivalence relation: The relation on finite state automata, where

\[ A \sim A' \]

if and only if there is a bisimulation between them. There’s no need to give a formal proof, just explain in English why you think the given relation has the required properties.

When programming we sometimes want to have a notion that two objects are equivalent for some equivalence relation. For example in Java the standard class `java.lang.Object` contains an instance method `equals()` which is designed to model the notion that two objects may be equivalent. See *Java: Just in Time* for more details in Section 20.4. This can only work properly if the relation in question is an equivalence relation.

### 7.4 Partial orders

Many structures we consider carry an order on them which allows us to compare their elements. There is a standard way of axiomatizing those properties that make a relation into an order in a way that agrees with our intuitions.

#### 7.4.1 Posets

When comparing two elements \( s \) and \( s' \) we write

\[ s \leq s' \]

to state that \( s \) is less than, or equal to, \( s' \), and

\[ s' \geq s \]

to state that \( s' \) is greater than or equal to \( s \). We consider the two equivalent. We further write

\[ s < s' \]

if and only if

\[ s \leq s' \quad \text{and} \quad s \neq s'. \]

We are using \( \leq \) to give the infix notation for a binary relation on a set \( S \). In this section we study the properties a binary relation needs to have in order to be something that behaves as we expect a comparison to behave.
First of all we expect each element \( s \) of \( S \) to be less than or equal to itself, so we expect 
\[
s \leq s
\]
to hold for all \( s \in S \). This means that we expect \( \leq \) to be a reflexive relation. Secondly we expect to be able to combine comparisons, that is if 
\[
s \leq s' \quad \text{and} \quad s' \leq s''
\]
for elements \( s, s', s'' \) from \( S \) then we expect to be able to conclude that 
\[
s \leq s''.
\]
In other words, we expect \( \leq \) to be transitive.
These two properties together are sufficient to ensure that \( \leq \) has most of the properties we expect from a comparison (see Optional Exercise 35), but it is customary to demand more.
When we know that 
\[
s \leq s' \quad \text{and} \quad s' \leq s
\]
hold for elements \( s \) and \( s' \) of \( S \) we want to be able to conclude that 
\[
s = s'.
\]

**Definition 58: antisymmetric**

A binary relation \( R \) on a set \( S \) is **anti-symmetric** if for all elements \( s \) and \( s' \) from \( S \) we have that 
\[
(s, s') \in R \quad \text{and} \quad (s', s) \in R \quad \text{implies} \quad s = s'.
\]

We cannot describe this property without having a binary equality predicate, say \( E \). If we use that then the first order formula expressing anti-symmetry is
\[
\forall x, \forall y. (((R(x, y) \land R(y, x)) \rightarrow E(x, y)).
\]
We may describe this property by noting that it says that 
\[
R \cap R^{\text{op}} \subseteq I_S.
\]
Anti-symmetry means that two elements cannot be less than or equal to each other unless they are equal. This property is the opposite of demanding symmetry of a relation, see Exercise 167, which explains the name.
In other words the only way we could have a two-sided arrow in the picture of an anti-symmetric relation is for that arrow to be a loop.\(^{27}\)

**Exercise 167.** Show that a symmetric reflexive relation on a set \( S \) is anti-symmetric if and only if it is the identity on \( S \).

Note that unlike for the other properties of binary relations studied in these notes, there is no such thing as an ‘anti-symmetric closure’ of a relation. For the other properties it is possible to add elements to the relation to obtain the desired property. But to make a relation anti-symmetric we would have to remove one of \((s, s')\) or \((s', s)\) if both are elements of the relation, and there’s no sensible way to decide which one to take away. One could remove both, but then one constructs something that is a long way removed from the starting relation.

\(^{27}\)But we don’t draw loops with arrows at both ends!
Exercise 168. Show that the transitive closure of the reflexive closure of an anti-symmetric relation is not necessarily anti-symmetric. Hence we cannot generate partial orders from relations in the way we can do with equivalence relations. *Hint: The smallest counterexample is a binary relation on a set with three elements.*

Now we have all the properties required to define a partial order.

**Definition 59: partial order**

A partial order on a set $S$ is a binary relation which is reflexive, antisymmetric and transitive. A set together with a partial order is known as a partially ordered set, or short, a poset.

We often write $(S, \leq)$ for the poset consisting of the set $S$ with the partial order $\leq$.

**Example 7.55.** The standard definition for $\leq$ on sets of numbers between $\mathbb{N}$ and $\mathbb{R}$ are all partial orders.\(^{28}\)

ALL these orders on sets of numbers satisfy an extra condition, see Definition 60, and you may find that you are assuming properties because you are guided by the examples you already know. Nonetheless we use the symbol $\leq$ for all partial orders in this section, so be careful not to make assumptions about such relations you make.

In order to check that a given relation on a small finite set is a partial order we may draw it as a directed graph, and then check reflexivity and transitivity as before. To check anti-symmetry we just have to make sure that there’s no connection with arrow tips on both ends. To put it differently, we make use of the transitivity of the relation when interpreting the diagram. It is customary, however, to draw a partial order on a small finite set as an undirected graph that is oriented on the page. Since a partial order is always reflexive one usually does not draw this part of the relation. Since a partial order is typically not symmetric (indeed, it has a property which is in a way the opposite, see Exercise 167), you might think that one should draw a directed graph, but instead the convention is to use the orientation of the paper to provide direction: Whenever two elements are connected by a line then the one that sits lower on the page is considered to be less than or equal to the other.\(^{29}\) Furthermore, in order to avoid drawing superfluous lines, the assumption is that if $a$ is connected to $b$ which is higher up on the page, and $b$ is connected to $c$, which is higher up again on the page, then the relation connects $a$ and $c$ even if no connection is drawn between the two. The resulting graph is known as the Hasse diagram of the given partial order.

\(^{28}\)Moreover, the operations on these sets of numbers, namely multiplication and addition, have specific properties which mean they are compatible with that order.

\(^{29}\)It should go without saying that this means that we have to draw our Hasse diagrams in such a way that two elements on the same level are never connected.
**Example 7.56.** Consider the partial order on \( \{a, b, c, d, e, f\} \) which is the transitive reflexive closure of

\[
\{(a, b), (b, c), (a, e), (e, f)\}
\]

Previously we would have drawn this relation as follows.

But for relations known to be partial orders we draw the corresponding Hasse diagram as explained in the preceding paragraph, which for the given poset looks like this:

The picture tells us (because of the orientation on the page) that \( a \) is less than or equal to \( b \) which in turn is less than or equal to \( c \), and that \( a \) is also less than or equal to \( e \), which in turn is less than or equal to \( f \). The element \( d \) is not comparable with any element other than itself.

**Example 7.57.** If we go back to the example from page 373, and form the transitive closure of the reflexive closure of

\[
\{(0, 1), (1, 2), (2, 3), (3, 4)\}
\]

then the Hasse diagram of this partial order is simply
Note that this is a total order (see Definition 60). In the Hasse diagram of a totally ordered set all the elements are lined up.

**Example 7.58.** Consider the following partial order on the natural numbers without 0, \( \mathbb{N} \setminus \{0\} \). We set
\[
m \leq n \quad \text{if and only if} \quad m \text{ divides } n.
\]
We show that this is a partial order by checking the properties required.  
- Clearly the relation defined above is reflexive since every natural number other than 0 divides itself.
- The relation is anti-symmetric since, given two natural non-zero numbers \( m \) and \( n \),
\[
m \text{ divides } n \quad \text{and} \quad n \text{ divides } m
\]
implies \( m = n \).
- The relation is transitive—you can find a proof in Exercise 23 (c).

Drawing a picture of (part of) this partial order, for the numbers from 1 to 10, gives a bit of an idea of what it looks like.
Example 7.59. Classes in Java are given a partial order by inheritance. You can define that one class is less than or equal to another if and only if it is a subclass.

Example 7.60. Many rankings come close to being partial orders. But typically it is possible for two entities to be considered equal in ranking, and so they often are not anti-symmetric. For example, ranking tennis, golf or chess players by the points they have accumulated has this property, as does ranking students by their overall averages. In the case where such a relation is anti-symmetric it is a total order, compare Definition 60. See Optional Exercise 35 for a look at relations that only lack anti-symmetry to be partial orders.

Example 7.61. We can define something that is almost a partial order on finite state machines by setting

\[ A \leq A' \quad \text{iff} \quad \text{there is a simulation from } A \text{ to } A'. \]

This is reflexive since the identity relation is always a simulation. It is transitive since the relational composite of two simulations is a simulation (see Optional Exercise 30. It is not anti-symmetric since it is possible for two automata to have simulations going each way without the automata being identical. Optional Exercise 35 encourages you to think about relations that have these properties.

Example 7.62. Scheduling is typically performed based on information that Task $A$ has to be performed before Task $B$. This information has to be

- anti-symmetric, since otherwise it is impossible to schedule all tasks;
- transitive since if Task $A$ has to come before Task $B$, which in turn

---

Note that these properties are established above in examples or exercises but we repeat some of the arguments here for completeness’ sake.
has to happen before Task C can be performed, then Task A has to happen before Task C.

Hence the reflexive closure of such a relation is a partial order. If we have to schedule tasks in a linear fashion, maybe because there is only one person (or robot or machine) to carry out all the tasks, then a valid schedule is one where

Task A can only be scheduled when

all tasks \( B \) with \( B \prec A \) have been scheduled.

We identify a special property shared by the partial orders on sets of numbers, see Example 7.55. Note that the following notion is defined in Section 20.3 of *Java: Just in Time*.

**Definition 60: total order**

A **total order** on a set \( S \) is a partial order \( \leq \) with the property that for all \( s \) and \( s' \) in \( S \) we have

\[
s \leq s' \quad \text{or} \quad s' \leq s.
\]

In this situation we say that the poset \((S, \leq)\) is **totally ordered**.

Again we give the first order proposition for this condition, where we assume a binary relation symbol \( R \) for the partial order.

\[
\forall x. \forall y. (R(x, y) \lor R(y, x))
\]

In a total order every two elements are comparable, so we may think of them all as being lined up, the way we usually think of \( \mathbb{N} \) or \( \mathbb{R} \). When we do not have this extra property then you may find that (partial) orders don’t quite behave in the way you expect.

**Example 7.63.** Examples of totally ordered sets are \( \mathbb{N} \), \( \mathbb{Z} \), \( \mathbb{Q} \) and \( \mathbb{R} \) with the usual order.

**Example 7.64.** The partially ordered set from Example 7.57 shows a total order.

**CExercise 169.** Which of the following relations are partial orders? Try to sketch a Hasse diagram for each\(^\text{31}\), and determine whether it is total. Justify your answers.

(a) The relation on complex numbers where \( a + bi \) is less than or equal to \( a' + b'i \) if and only if \( a \leq a' \) and \( b \leq b' \).

(b) The relation on complex numbers where \( a + bi \) is less than or equal to \( a' + b'i \) if and only if \( a \leq a' \).

(c) The relation on \( \mathbb{N} \times \mathbb{N} \) where \( (m, n) \) is less than or equal to \( (m', n') \) if and only if \( m + n \leq m' + n' \).

(d) The relation on \( \mathbb{N} \times \mathbb{N} \) where \( (m, n) \) is less than or equal to \( (m', n') \)
if and only if

\[ m < m' \quad \text{or} \quad m = m' \quad \text{and} \quad n \leq n'. \]

(e) The relation on functions from \( \mathbb{N} \) to \( \mathbb{N} \) where \( f \) is less than or equal to \( g \) if and only if (compare Section 5.1)

\[ g \text{ dominates } f, \]

that is

for all \( n \in \mathbb{N} \) we have \( fn \leq gn \).

This is known as the \textit{pointwise order} on \( \text{Fun}(\mathbb{N}, \mathbb{N}) \) since we look at each input point.

(f) The subset relation on the powerset of a set \( X \).

(g) The relation on strings over a set \( S \) where string \( l \) is less then or equal to string \( l' \) if and only if \( l \) is a \textit{prefix} of \( l' \)—in other words, if the beginning of \( l' \) is \( l \).

(h) Assume we have a poset \((S, \leq)\). Then we obtain an order on the set of strings over the set \( S \), known as the \textit{lexicographic order} by setting\(^\text{12}\)

\[ s_1 s_2 \cdots s_m \preceq s'_1 s'_2 \cdots s'_n \quad \text{if and only if} \]

for \( i \) smallest number in \( \mathbb{N} \) with \( s_i \neq s'_i \)

we have \( s_i < s'_i \) or \( s_i \) not defined.

Show that the relation defined in this way is indeed reflexive, antisymmetric and transitive.

(i) The relation on the set of first year students in the School where student \( A \) is related to student \( B \) if and only if student \( B \) is taller than student \( A \).

(j) The relation on the set of first year students in the School where student \( A \) is related to student \( B \) if and only if the registration number of student \( A \) is less than or equal to that of student \( B \).

(k) The relation on all valid \textsf{Java} programs where programme \( P \) is less than or equal to program \( Q \) if and only if as a string, in the lexicographic order (also known as the ‘dictionary order’) program \( P \) comes before program \( Q \).

(l) The following relation on the set \{\( a, b, c, d, e \)\}:

\[ \{(a, a), (b, b), (c, c), (d, d), (e, e)\}. \]

(m) The reflexive closure of the following relation on \{\( a, b, c, d, e \)\}:

\[ \{(a, b), (c, d)\}. \]
(n) The reflexive transitive closure of the following relation on the same set.
\{ (a, b), (b, c), (c, d), (d, e) \}.

(o) The reflexive transitive closure of the following relation on the same set.
\{ (a, b), (b, a), (b, c) \}.

(p) The subset relation on the powerset of the set \{ 0, 1, 2 \}.

*In parts (a) to (e) you have an existing relation \( \leq \) on a set of numbers \( \mathbb{N} \) or \( \mathbb{R} \), and a new relation is defined based on that. You may find it less confusing to use a new symbol for the new relation, maybe \( \sqsubseteq \) or \( \preceq \) as I do in part (h).*

---

**Example 7.65.** We show how to define a partial order on \( \text{FBTrees}_S \), where \( S \) is an arbitrary set. We would like to define\(^{33}\)

**Base cases** \( \leq \).
For all \( s \in S \), we have \( \text{tree}_s \leq \text{tree}_s \).

For all \( s \in S, t, t' \in \text{FBTrees}_S \), \( \text{tree}_s \leq \text{tree}_{s(t, t')} \).

**Step case** \( \leq \). If \( t \leq t'' \) and \( t' \leq t''' \) then \( \text{tree}_s(t, t') \leq \text{tree}_s(t'', t''') \).

This relates two trees if and only if the second arises from the first by repeatedly extending the tree, that is

- pick a leaf with label \( s \) (that is a subtree of the form \( \text{tree}_s \)) and
- replace that leaf by a tree whose root label is \( s \), that is, a tree of the form \( \text{tree}_s(t, t') \).

So, for example, we have that the tree on the left is less than or equal to the one on the right.

\[ 3 \]
\[ 2 \]
\[ 2 \]
\[ 2 \]
\[ 1 \]
\[ 3 \]
\[ 3 \]
\[ 1 \]
\[ 1 \]

Alternatively we can think of the relation \( \leq \) as relating a tree \( t \) to a tree \( t' \) if and only if we can obtain the first tree from the second tree by cutting off some of the branches.

We can show that this is a partial order. For reflexivity we have an inductive proof.

---

\(^{31}\)For infinite or large sets just try to get a general idea what such a diagram might look like and draw a part of it.

\(^{32}\)Any dictionary gives you the words in this order. Also, the Linux list function presents the contents of some directory in this order.

419
Base cases ≤. For \( s \in S \) we know that \( \text{tree}_s \leq \text{tree}_s \) by the first base case of \( \leq \).

**Ind hyp.** We know that that \( t \leq t \) and \( t' \leq t' \) for some full binary trees \( t \) and \( t' \) over \( S \).

Step case ≤. From the induction hypothesis we may use the step case of \( \leq \) to deduce that for every \( s \in S \) we have

\[
\text{tree}_s(t, t') \leq \text{tree}_s(t, t').
\]

We can show anti-symmetry by a second induction proof.

Base cases ≤. For \( t, t' \in \text{FBTrees}_S \) if \( t \leq t' \) by one of the base cases of the definition then it must be the case that we can find \( s \in S \) such that \( t = \text{tree}_s \).

In order for \( t' \leq t \) to also hold it must be the case, since we know that \( t = \text{tree}_s \), that this also occurs by the first base case of the definition of \( \leq \), and so we must have that there is \( s' \in S \) with \( t' = \text{tree}_{s'} \), but But if they are related by the base case of the definition of \( \leq \) then it must also be the case that \( s = s' \), and so

\[
t = \text{tree}_s = \text{tree}_{s'} = t'.
\]

**Ind hyp.** If \( t \leq t' \) and \( t' \leq t \) then \( t = t' \).

Step case ≤. The only remaining case is that

\[
\text{tree}_s(t, t') \leq \text{tree}_s(t', t'') \quad \text{and} \quad \text{tree}_s(t', t''') \leq \text{tree}_s(t, t')
\]

both by the step case, and so

\[
t \leq t' \quad \text{and} \quad t'' \leq t''', \quad \text{and} \quad t' \leq t \quad \text{and} \quad t''' \leq t''
\]

and by the induction hypothesis

\[
t = t' \quad \text{and} \quad t'' = t'''.
\]

Note that this is a more complicated induction proof than others contained in these notes. This is required to ensure that all possible cases are covered. The proof of transitivity is a similar induction proof, see the following exercise.

**Exercise 170.** This exercise studies in more detail the partial order on trees described in the preceding example.

\[33\text{Note that there are many base cases here, one for each tree!}\]
(a) Rewrite the recursive definition of this partial order to define a set $R_\leq \subseteq \text{FBTrees}_S \times \text{FBTrees}_S$.

(b) Show by induction that for $(t, t') \in R_\leq$ we have that the roots of the two trees have the same label.

(c) Show that we have for $t, t' \in \text{FBTrees}_S$ and the partial order $\leq$ from Example 7.65 that $t \leq t'$ if and only if $(t, t') \in R_\leq$.

(d) Show that $\leq$ as defined in Example 7.65 is transitive.

(e) Write a program that takes two objects of class BTree and returns true if and only if the first is less than or equal to the second in this order.

**Example 7.66.** We can define a partial order on the set Lists$_S$ of lists over a set $S$, where one list is less than or equal to another if and only if the second list is an extension of the first list, that is, it arises from the first list by adding more elements (from the left as usual). Note that while this has many similarities to the previous example there are subtle differences.

So, for example, if $S = \mathbb{N}$ we have

$[4, 3, 2, 1] \leq [7, 9, 3, 4, 4, 3, 2, 1],$

because we have that $[7, 9, 3, 4, 4, 3, 2, 1]$ may be obtained from $[4, 3, 2, 1]$ by adding the elements 4, 3, 9, and 7 in that order. On the other hand, the list $[4, 3, 2]$ is not less than or equal to $[4, 3, 2, 1]$ because the two lists don't start out in the same way.

We can begin to sketch a Hasse diagram for this poset.
We can give a recursive definition for the set of all pairs that belong to this relation \( R \leq \) on \( \text{Lists}_S \). In other words, we recursively define

\[
R \leq \subseteq \text{Lists}_S \times \text{Lists}_S
\]
similar to the way we defined the subset of \( \mathbb{N} \) that consists of all even numbers (Example 6.42), or of all powers of 2 (Example 6.43).

**Base cases** \( R \leq \).

- \( l \in \text{Lists}_S \) implies \((l, l) \in R \leq \).

**Step case** \( R \leq \).

- \((l, l') \in R \leq \), \( s \in S \) implies \((l, s : l') \in R \leq \).

If we are allowed to apply the reflexive transitive closure operation afterwards we can get away with a simpler definition of a relation, say \( \hat{R} \):

- For \( l \in \text{Lists}_S \) and \( s \in S \) we have \((l, s : l) \in \hat{R} \).

The relation \( R \) is the reflexive transitive closure of the relation \( \hat{R} \).

Alternatively we can write a method which takes as input two objects of class List and returns true if and only if the first list is less than or equal to the second list. (You may want to compare this to Example 132.)

```java
public static boolean lessthan (List l1, List l2)
{
    if (l1 == null)
        return true;
    else
    {
        if (l2 == null)
            return false;
        else
        {
            if (l1.value == l2.value)
                return lessthan (l1.next, l2.next) || lessthan (l1,l2.next);
            else
                return lessthan(l1, l2.next);
        }
    }
}
```

Can you see why we need the disjunction clause after the third if statement? If not, try to take the examples from above through this method.
In some ways it is easier to recursively define the relation than to define this method. In other words, defining an algorithm that checks whether two lists are in the relation is harder than defining the relation. Indeed, finding decision procedures (which you encounter in COMP11212) can be hard for properties that can be described in a simple manner. —if we want to know whether a given pair

Exercise 171. Show the following for the relation \( R \leq \) defined in the preceding example:

(a) The relation \( R \leq \) is reflexive.
(b) If \((l, l') \in R \leq\) then \( \text{len } l \leq \text{len } l'\).
(c) If \((l, l') \in R \leq\) and \( \text{len } l = \text{len } l' \) then \( l = l'\).
(d) The given relation is anti-symmetric.
(e) Show that if we have \( l \) and \( l' \) in Lists\(_S\), and \( n \in \mathbb{N} \), and \( s_1, s_2, \ldots, s_n \) in \( S \), with

\[
l' = s_n : (s_{n-1} : (\ldots (s_1 : l)))
\]

then

\[
(l, l') \in R \leq.
\]
(f) Show the converse of the previous part, that is, given \((l, l') \in R \leq\) then we can find elements in \( S \) to satisfy the given condition.
(g) Show that the given relation is transitive.

For any given part you may need to use a previous part to give the desired proof.

Note that when programming, we sometimes come across ways of comparing elements which are not partial orders. For example, the CompareTo method for many classes in Java typically implements a comparison that is not antisymmetric. It is an instance of a formal notion explored in the following (optional) exercise.

Optional Exercise 35. A pre-order \( \preceq \) is a binary relation on a set \( S \) which is reflexive and transitive. For such a pre-order it makes sense to define a relation \( \approx \) by, for \( s \) and \( s' \) in \( S \),

\[
s \approx s' \quad \text{if and only if} \quad s \preceq s' \text{ and } s' \preceq s.
\]

(a) Can you think of an example of a pre-order? Having an example at hand my help you with the following parts.
(b) Show that \( \approx \) is an equivalence relation.

\footnote{However, one can take that recursive definition of the relation and turn it into the decision procedure given following a specific set of rules! This idea takes us beyond the scope of this course unit.}
(c) Use $\leq$ to define a partial order on $S/\approx$ and show that your relation is indeed a partial order. Conclude that for every pre-order there is a corresponding partial order on a related set.

(d) The `compareTo()` method in Java for many classes is concerned with a pre-order, rather than a partial one. If you want to think more about pre-orders you could verify this claim, and think about why the method was designed in this way.

(e) Another pre-order is given by the relation between sets where $S$ is related to $T$ if and only if the size of $S$ is at most as bit as that of $T$. Why is this not a partial order?

**Exercise 172.** Show that the set of all functions

$$\text{Fun}(X, N),$$

where

- $X$ is any set, and
- $N$ is a set of numbers between $\mathbb{N}$ and $\mathbb{R}$

carries a partial order given by the ‘is dominated by’ relation from Section 5.1, which is known as the *pointwise order* (compare Exercise 169 (e)).

**Exercise 173.** Show that if $(P, \leq)$ is a poset then we obtain another partial order by the following means.

(a) By setting

$$p \subseteq p' \quad \text{if and only if} \quad p' \leq p.$$

This is known as the *opposite (partial) order* of the poset, since $\subseteq$ is $\leq^{\text{op}}$. It means turning the poset upside-down.

(b) By taking a subset $Q$ of $P$ and defining, for $q$ and $q'$ in $Q$,

$$q \subseteq q' \quad \text{if and only if} \quad q \leq q' \text{ as elements of } P.$$

This is known as the *subset (partial) order* on $Q$.

**Optional Exercise 36.** The total orders on $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ you are used to are very well behaved. For example, for numbers $m$, $n$, $m'$ and $n'$ we have

$$m \leq m' \text{ and } n \leq n' \quad \text{implies} \quad m + n \leq m' + n'.$$

(a) Prove this property for $\mathbb{N}$ using induction, and the recursive definitions of $\leq$ and $+$. 

424
(b) Prove this property for \( \mathbb{Z} \) and \( \mathbb{Q} \) using their formal definitions from the previous section.

(c) Can you think of a similar property that holds for multiplication? *Hint: Have a look at Fact 7/*

(d) There is no standard partial order on the complex numbers \( \mathbb{C} \). While it is possible to define a number of partial orders on this set (compare Exercise 169) most of these are not total,\(^{35}\) nor will the operations be well-behaved with respect to these partial orders. Conduct some experiments by defining orders on \( \mathbb{C} \) and exploring their properties.

### 7.4.2 Maximal, minimal, greatest, least

Having a partial order on a set allows us to find elements with distinctive properties.

**Definition 61: maximal/minimal**

An element \( p \) of a poset \((P, \leq)\) is a **maximal element** of \( P \) if and only if for all \( p' \in P \),

\[
\text{if } p \leq p' \text{ then } p = p'.
\]

An element \( p \) of a poset \((P, \leq)\) is a **minimal element** of \( P \) if and only if for all \( p' \in P \) we have

\[
\text{if } p' \leq p \text{ then } p' = p.
\]

The propositional formula in the predicate calculus that expresses this property requires a constant \( p \) to be a maximal element for a partial order, a binary relation symbol \( R \) for the partial order, and a binary relation \( E \) for equality. It looks as follows.

\[
\forall x. (R(p, x) \rightarrow E(p, x))
\]

The way to understand this definition is to note that for a maximal element \( p \) of a poset there cannot be another element above it (because any element above has to be equal to \( p \)). Analogously, a minimal element cannot have another element below it.

Note that a poset can have more than one maximal or minimal element, see the following example.

**Example 7.67.** If we go back to our running example from above,

\[
\begin{align*}
&c \\
&\quad \downarrow \\
&b \\
&\quad \downarrow \\
&a
\end{align*}
\begin{align*}
&f \\
&\quad \downarrow \\
&e \\
&\quad \downarrow \\
&d
\end{align*}
\]

\(^{35}\)What would a total order on the set \( \mathbb{C} \) look like?

425
we can see that the maximal elements are $c$, $f$ and $d$, and that there are two minimal elements, namely $a$ and $d$.

This example illustrates that a maximal element does not have to be above all the other elements (for example, $f$ is maximal but not above $c$, $b$ or $d$, and $d$ is maximal but not above anything other than itself).

**Example 7.68.** By Exercise 173 we may turn our given poset upside-down to obtain another poset:

In this poset there are three minimal elements, $c$ and $f$ and $d$, and two maximal elements, $a$ and $d$.

It is immediately clear from the definition that when we turn a poset upside-down then maximal elements become minimal, and vice versa. We look at another concept which is more restricted than being minimal or maximal.
An element \( p \) is the greatest element of a poset \( (P, \leq) \) if and only if it is the case that

\[
\text{for all } p' \in P \quad \text{we have} \quad p' \leq p.
\]

The greatest element is usually called \( \top \).

An element \( p \) is the least element of a poset \( (P, \leq) \) if and only if it is the case that

\[
\text{for all } p' \in P \quad \text{we have} \quad p \leq p'.
\]

The least element is usually called \( \bot \).

Using the same convention as before the proposition corresponding to a parameter \( p \) describing a largest element for a partial order given by a binary relation symbol \( R \) is

\[
\forall x. R(x, p).
\]

**Example 7.69.** Our original example poset has no greatest or least element. If we remove the element \( d \) from the set we obtain the following.

\[
\begin{array}{c}
\text{c} \\
\downarrow \\
\text{b} \\
\downarrow \\
\text{a} \\
\end{array}
\begin{array}{c}
\text{f} \\
\downarrow \\
\text{e} \\
\end{array}
\]

This poset has no greatest element; a least element exists, namely \( a \). Note that this is also a minimal element, and that there is only one minimal element. See the following exercises for this and other important properties of least and greatest elements.

**Example 7.70.** Consider the set \( \mathbb{N} \setminus \{0\} \) where

\[
m \leq n \quad \text{if and only if} \quad m \text{ divides } n,
\]

compare Example 7.58.

Since 1 divides every number it is the least element. This means that it is the only minimal element.

Since no number is divided by all natural numbers there is no largest element. But there are no maximal elements either: For every element
For $n \in \mathbb{N} \setminus \{0\}$ we have that $n$ divides $2n$, and so for any given $n \in \mathbb{N} \setminus \{0\}$ we can always find a larger element.

**Example 7.71.** We take the previous example but restrict to the set $\mathbb{N} \setminus \{0,1\}$ where we again define

$$m \leq n \quad \text{if and only if} \quad m \text{ divides } n,$$

compare Example 7.58. Compared to that example we have removed the number 1 from consideration.

![Hasse diagram](image)

There is no number in $\mathbb{N} \setminus \{0,1\}$ that divides all numbers, so there is no least element. There are a lot of minimal elements however: In order for an element $n$ to be minimal we must have that $m \leq n$ implies $m = n$; in other words, we are looking for numbers $n$ with the property that for all $m \in \mathbb{N} \setminus \{0,1\}$,

$$m \text{ divides } n \quad \text{implies} \quad m = n.$$

This is the case exactly when $n$ is a prime number, and so all prime numbers are minimal.

Since no number is divided by all natural numbers there is no largest element. But there are no maximal elements either: As in the previous example we have that for every element $n \in \mathbb{N} \setminus \{0\}$ we have that $n$ divides $2n$, and so we can always find a larger element.

**Example 7.72.** For the partial order on $\text{Lists}_S$ from Example 7.66 we have a least element, namely the empty list $[]$. There are no maximal elements since given an arbitrary list we can always make it bigger by adding another element.

**Exercise 174.** For the following posets, try to draw a Hasse diagram and determine any maximal, minimal, least and greatest elements.

(a) The real numbers with the usual order.

(b) The natural numbers with the usual order.

(c) The negative integers with the usual order.

(d) For the partial order on $\text{FBTrees}_S$ from Example 7.65.
(e) The powerset of a set $X$ with subset inclusion as the order.

(f) The set $\text{Fun}(\mathbb{N}, \mathbb{N})$ of functions from $\mathbb{N}$ to $\mathbb{N}$ with the pointwise order, given by:

$$f \leq g \quad \text{if and only if} \quad \text{for all } n \in \mathbb{N} \text{ we have } fn \leq gn.$$ 

(g) The following relation on $\{a, b, c, d, e\}$.

$$\{(a, a), (b, b), (c, c), (d, d), (e, e)\}.$$ 

(h) The reflexive closure of the following relation on $\{a, b, c, d, e\}$:

$$\{(a, b), (c, d)\}.$$ 

(i) The reflexive transitive closure of the following relation on that set.

$$\{(a, b), (a, c), (b, d), (b, e), (c, d), (c, e)\}.$$ 

(j) The reflexive transitive closure of the following relation on that set.

$$\{(a, b), (b, c), (c, d), (d, e)\}.$$ 

(k) For your programme of study, and the set of compulsory COMP units on that programme (through years 1 to 3), the partial order which is the transitive reflexive closure of the ‘is a prerequisite of’ relation.

(l) For Chapter 6, take all operations defined in that chapter (including exercises) for $\mathbb{N}$ and for lists, and use the transitive reflexive closure of the ‘is required to define’ relation (for example, the addition operation is required to define multiplication of natural numbers).

**Exercise 175.** Greatest and least elements have special properties.

(a) Show that there is at most one greatest, and at most one least, element in each poset.

(b) Show that if $p$ is the greatest element of a poset then it is maximal. Similarly, show that if $p$ is the least element of a poset then it is minimal.

(c) Show that if a poset has a greatest element then it has exactly one maximal element. Similarly show that if a poset has a least element then it has exactly one minimal element.

(d) Show that if $(P, \leq)$ is a totally ordered poset then every maximal element is a greatest element, and every minimal element is a least element. Conclude that a totally ordered set can have at most one maximal, and one minimal, element.
7.4.3 Upper and lower bounds

When looking at posets we often care about whether a collection of elements can be safely ‘overestimated’ (or ‘underestimated’) by using a single element. For example, when we are looking at the complexity of a problem we may be interested in the fact that it is ‘as most as complicated as some function $f$’—that means that there is an algorithm with complexity function less than or equal to $f$. Or maybe we have a group of algorithms or programs we would like to discuss in one go, and we can state that the complexity function for each of these lies below some ‘upper bound’ given by some function $f$.

**Definition 63**: upper/lower bound

Let $S$ be a subset of a poset $(P, \leq)$. We say that an element $p$ of $P$ is an **upper bound** for $S$ if and only if it is the case that

$$\forall p' \in S \quad p' \leq p.$$ 

We say that an element $p$ of $P$ is a **lower bound** for $S$ if and only if

$$\forall p' \in S \quad p \leq p'.$$

Again we give a first order proposition to describe this definition. Assume we have

- a binary predicate symbol $R$ for the partial order,
- a unary predicate $S$ that describes the elements of the set $S$, so $S(x)$ is interpreted as 1 if and only if $x$ is in $S$,

then the proposition establishing $p$ as an upper bound of $S$ is

$$\forall y. (S(y) \rightarrow R(y, p)).$$

The example poset we have been using so far is too simple to contain many interesting upper or lower bounds. For the version of the example poset that appears in Example 7.69, the only interesting observation is that $a$ is the only lower bound for the sets $\{c, f\}$, $\{b, f\}$, $\{c, e\}$ and $\{b, e\}$.

**Example 7.73.** Consider the poset drawn below.

![Diagram of a poset]

39 Note that $p$ need not be an element of $S$.
40 Again note that $p$ need not be an element of $S$. 
What are the upper bounds for the set \( \{b, e\} \)? They are the elements which are above both, \( b \) and \( e \).

![Diagram showing upper bounds of \( \{b, e\} \)]

We can see that the upper bounds are \( c \) and \( g \).

What are the upper bounds of \( \{c, f\} \)?

![Diagram showing upper bounds of \( \{c, f\} \)]

The element \( g \) is the only upper bound of the sets \( \{c, f\} \) (and also of \( \{b, f\} \)).

**Example 7.74.** In Java we can order classes by the subclass relation. For this partial order two classes have a common upper bound if and only if there is a class which is a superclass for both.

**Exercise 176.** For the poset from Example 7.73, find all lower bounds for the sets \( \{b, e\} \), \( \{g, b\} \), \( \{g, e\} \), \( \{c, f\} \), \( \{b, f\} \).

Sometimes we can find a ‘best’ upper bound for a subset of a poset, that is, amongst all the upper bounds there may be a least one.

**Definition 64: least upper/greatest lower bound**

Let \( S \) be a subset of the poset \( (P, \leq) \). We say that an element \( p \) of \( P \) is the 
\(^{41}\) least upper bound (or supremum) for \( S \) if and only if

- \( p \) is an upper bound for \( S \) and
- if \( p' \) is an upper bound for \( S \) then \( p \leq p' \).
The least upper bound of two elements \( p \) and \( p' \) is often written as \( p \lor p' \). We say that an element \( p \) of \( P \) is the greatest lower bound (or infimum) for \( S \) if and only if

- \( p \) is a lower bound for \( S \) and
- if \( p' \) is a lower bound for \( S \) then \( p' \leq p \).

The greatest lower bound of two elements \( p \) and \( p' \) is often written as \( p \land p' \).

This statement is fairly complicated if we look at the number of quantifiers involved when we unravel the statements from above. In an appropriate formal system for \( p \) to be the least upper bound of \( S \) we need

\[
\forall y. (S(y) \rightarrow R(y, p)) \\
\land \\
\forall y. ((\forall z. S(z) \rightarrow R(z, y)) \rightarrow R(p, y)).
\]

You can see that by building up the definitions slowly we have a final definition which looks simpler. But taking it apart reveals that the concept is quite complicated, as can be seen from looking at the corresponding proposition above.

Example 7.75. Going back to the example from above: The set \( \{b, e\} \) has two upper bounds, namely \( c \) and \( g \). This set has a least element, namely \( c \), and so \( c \) is the least upper bound of \( \{b, e\} \).

An alternative way of looking at least upper bounds is the following: We can form a new set from \( S \), namely

\[
\{p' \in P \mid p' \text{ upper bound for } S\},
\]

and then

\( p \) is a least upper bound for \( S \)

if and only

\( p \) is the least element of \( \{p' \in P \mid p' \text{ upper bound for } S\} \).

In particular,

\[
\{p' \in P \mid p' \text{ upper bound for } S\}
\]

must have a least element for \( S \) to have a least upper bound.

---

41 The definite article is justified by Exercise 177.

42 We need a one-placed predicate symbol for ‘is an element of \( S \)’ and a binary predicate symbol \( R \) for the partial order.

432
**Example 7.76.** If, on the other hand, we change our underlying poset slightly to

```
  a
   
  b  e  f
   
  c
```

then $c$, $f$ and $g$ are all upper bounds of $\{b, e\}$, but none of them is the least upper bound since the set

$$\{c, f, g\}$$

has no least element.

**Example 7.77.** We return to another of our running examples. Consider once again the set $\mathbb{N} \setminus \{0\}$ with the partial order given by

$$m \leq n \quad \text{if and only if} \quad m \text{ divides } n,$$

cmpare Examples 7.58 and 7.70. We repeat the Hasse diagram.

```
  8   12
 /|
/  
4  6  9  10  
/|
/  
2  3  5  7  11  
/|
/  
1
```

If we have two numbers $m$ and $n$ then their lower bounds are the numbers which are below both of them, that is, all those numbers which divide both, $m$ and $n$. For example, for

$$12 \quad \text{and} \quad 18$$

these numbers are

$$\{1, 2, 3\}.$$
The set of common divisors of two numbers \( m \) and \( n \) always has a largest element, namely the greatest common divisor of \( m \) and \( n \), compare Example 6.41. Hence for every two elements there is a greatest lower bound, given by their greatest common divisor. Similarly, given two numbers \( m \) and \( n \), their upper bounds are those numbers which have both of them as divisors, that is, all the common multiples of the two. For example, for

\[
6 \quad \text{and} \quad 4
\]

these are all the numbers that are multiples of both,\(^{43}\) 4 and 3.

\[
\{12, 24, 36, \ldots \}
\]

which means they are all the multiples of 12,

\[
\{12n \in \mathbb{N} \mid n \in \mathbb{N}\}
\]

This set always has a smallest element, in this case 12. In general, the least upper bound of two numbers \( m \) and \( n \) is their smallest common multiple.

**Example 7.78.** If we return to the partial order on the set \( \text{Lists}_S \) of lists over a set \( S \) given in Exercise 7.66 we note the following:

Given any two lists, their greatest lower bound is the longest list they both have in common, so for

\[
[4, 3, 2, 1] \quad \text{and} \quad [5, 4, 2, 1],
\]

the greatest lower bound is

\[
[2, 1].
\]

Once those two elements have been added to the list the paths of getting to the two lists diverge. Note that there is always such a largest list (it might be the empty list), so for every pair of element a greatest lower bound exists. If you look at the Hasse diagram for this poset, which is drawn in Example 7.66 you can see that this looks like a tree. To find the greatest lower bound for two lists, you go down the branches of the tree which contain them until these branches come together.

On the other hand if we look at least upper bounds for two lists we find that they almost never exist. This is because two lists which are not related by the order do not have any upper bounds. For example, an upper bound for both

\[
[2, 1] \quad \text{and} \quad [3, 1]
\]

would have to be obtainable by adding elements to both these lists, and clearly this cannot happen. Hence the only case in which any upper

\(^{43}\)You may want to convince yourself that all numbers that are multiples of both, 4 and 3, are also multiples of both, 6 and 4.
bounds exist is in the case where

\[ l \leq l', \]

in which case \( l' \) is the least upper bound for the two.

**Exercise 177.** Show that every subset of a poset \( P \) has at most one least upper, or greatest lower, bound in \( P \).

So least upper (greatest lower) bounds of sets exist, or don’t, but where they exist they are unique.

**Exercise 178.** This exercise is concerned with calculating greatest lower and least upper bounds. Note that all the settings described here also appear in Exercise 169 and that you may want to solve the corresponding part of that exercise first.

(a) For the natural numbers with the usual order, how would you describe the greatest lower, and least upper, bounds for sets of the form \( \{m, n\} \)? Can you extend this to all finite subsets of \( \mathbb{N} \)? Do you need finiteness? Try to extend your result to all subsets of \( \mathbb{N} \) as far as possible.

(b) For the powerset of the set \( \{0, 1, 2\} \) how does one calculate the greatest lower, and least upper, bound of a subset? *Hint: You may want to draw the Hasse diagram. What are the greatest lower and least upper bounds of \( \{\{0, 1\}, \{1, 2\}\} \)

(c) Consider the powerset \( \mathcal{P}X \) of a set \( X \), with subset inclusion providing a partial order. Show that for every subset \( S \) of \( \mathcal{P}X \) we have

- The least upper bound of \( S \) exists and is given by
  \[
  \bigcup S = \bigcup \{S \subseteq X \mid S \in S\} = \{x \in X \mid \exists S \in S. x \in S\}.
  \]

- The greatest lower bound \( S \) exists and is given by
  \[
  \bigcap S = \bigcap \{S \subseteq X \mid S \in S\} = \{x \in X \mid \forall S \in S. x \in S\}.
  \]

(d) Consider the set of strings made from symbols from a set \( S \). Consider the partial order where a string is less than or equal to another if it is a prefix of the latter. What are the greatest lower, and least upper, bounds of sets of strings (where they exist)? *Hint: You have already seen a partial Hasse diagram for this set. Start from there by looking at the two-element subsets.

(e) Consider the set \( \text{FBTrees}_S \) of full binary trees over a set \( S \) with the partial order from Example 7.65. When do the least upper, and greatest lower, bounds of two trees exist and what are they? For this part it is sufficient to describe your answer in English and to give an example.
Exercise 179. Consider the following relation for the set \( \text{FBTrees}_\mathbb{N} \) of full binary trees over the set of labels \( \mathbb{N} \). Recall the function \( k_1 \) from Exercise 118 which maps a natural number to the number 1. Also recall the map function from the same exercise, which has the property that

\[
\text{(map } k_1 \text{)}t
\]

replaces every label in the tree \( t \) by the number 1. Finally recall the partial order \( \leq \) on \( \text{FBTrees}_S \) described in Example 7.65. We use this to define a new partial order on the same set.

For two trees \( t \) and \( t' \) we set

\[
t \sqsubseteq t'
\]

if and only if

\[
\text{(map } k_1 \text{)}t \leq \text{(map } k_1 \text{)}t'.
\]

(a) Try to describe in English under which circumstances we have that \( t \sqsubseteq t' \).

(b) Modify the code for the less_than method from Example 7.65 in such a way that it implements this new relation.

(c) Is this relation a partial order?

(d) For the set \( S = \{1\} \), consider the partial order \( \leq \) on \( \text{FBTrees}_S \) used above. Describe the greatest lower bound of two elements of this set in English.

(e) For the same poset as in the previous part describe the least upper bound of two elements in English.

Exercise 180. Consider the following relation on the set of partial functions from a set \( S \) to a set \( T \):

\[
f \preceq g \quad \text{if and only if} \quad \text{dom } f \subseteq \text{dom } g \text{ and } s \in \text{dom } f \text{ implies } fs = gs
\]

(a) Show that this relation is a partial order.

(b) What are the minimal elements of this poset? Is there a least element?

(c) What are the maximal elements of this poset? Is there a greatest element?

(d) Given two partial functions \( f \) and \( g \) from our set, under which circumstances is their greatest lower bound \( f \land g \) defined? Can you describe this partial function?
(e) Given two partial functions $f$ and $g$ from our set, under which circumstances is their least upper bound $f \lor g$ defined? Can you describe this partial function?

The mathematical discipline of order theory is concerned with studying partially ordered sets and their properties.
Chapter 8

Applications to Computer Science

The eventual aim of this chapter is to collect examples and exercises from computer science where mathematical techniques are routinely employed, but without stating explicitly which areas of mathematics might be involved. Currently a number of suitable exercises are located in various chapters. While they serve there to motivate some concepts and techniques, from a didactic point of view it would be good if students gained some experience in solving such problems without too many hints regarding what techniques to employ.

Examples are:

- The use of precise language, and the predicate calculus, in particular first order logic, to describe specifications for a computer program (some examples appear below).
- To verify the correctness of programs and protocols using techniques from logic, see for example COMP31111.
- Programming with logical formulae in the language Prolog which is taught in COMP24412.
- Solving optimization problems with tools from logic in COMP21111.
- The use of the expected value of a random variable to calculate the complexity of a given algorithm.
- The use of probabilities and expected values to evaluate risks.
- The notion of one function growing at most as fast as another to talk about complexity classes of problems or algorithms—this material is covered in COMP11212, COMP26120 and COMP36111.
- The use of the notion of the size of a set to answer questions from computer science, for example: Are there functions from \( \mathbb{N} \) to \( \mathbb{N} \) which cannot be implemented using a computer?
- The use of recursion to solve particular problems. This is covered in COMP16212, COMP26120 and COMP36111.
• The use of recursively defined function to work out the complexity of a particular algorithm using recurrence relations.

• The use of induction as a technique to prove properties of many entities that appear in various course units.

• Using relations to express properties of programming constructs, or of elements of some other domain of discourse.

• The use of vectors and matrices in particular in the context of graphics, which appears in COMP27112.

For the moment only a small number of exercises appear in this chapter. They are all concerned with analysing a particular problem with a view to creating a precise specification for the desired solution. Various exercises that appear in other chapters in these notes could be included in this chapter, but I did not want to postpone your tackling them until the end of term.

• **Chapter 2.** Various questions are concerned with the properties of Java operations, and with the properties of functions that assign user names.

• **Chapter 4.** Bayesian updating is a technique from Machine Learning and is covered in Exercises 76 to 78. Using expected values to calculate average complexities for algorithms is also a computer science topic, which is the topic of Exercise 93. We look a little bit at the reliability of hardware in Exercise 87, and where a system is built from components, in Exercise 98.

• **Chapter 6.** All the exercises asking you to produce code are applications of the mathematical idea of recursion to programming. Proofs by induction that a given recursively defined function has certain properties can be transplanted to recursive programs, and are required if you want to show that your code behaves as expected/specified.

• **Chapter 7.** We look at algorithms that calculate the powers of some number in modular arithmetic in Exercise 158. There are also exercises asking you to produce code for some problems in this chapter.

Much of the time programmers do not make the effort to precisely define what they want their code to do, but under some circumstances this is a very useful tool, for example

• when code already written shows unexpected behaviour or

• when one is trying to define the behaviour of a compiler of a programming language\(^1\) or

• for safety-critical applications, such as the control of a space or aircraft, when one would ideally have verification that the code implemented has particular properties.\(^2\)

\(^1\)You may be surprised by this, but a number of programming languages do not have precise definitions of their behaviour and different compilers (or interpreters) can behave differently for some programs.

\(^2\)A unit that looks at this issue is COMP31111, *Verified Development.*
**Exercise 181.** Imagine you are a programmer tasked with writing a program that takes as its input an array of natural numbers, and returns a sorted version of that array. The aim of this exercise is to specify what exactly this means.

For example, you would expect for the input array \( a \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
9 & 35 & 0 & 5 & 58 & 7 & 1 & 42 \\
\end{array}
\]

to produce the following output array, say \( b \):

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & 1 & 5 & 7 & 9 & 35 & 42 & 56 \\
\end{array}
\]

For the conditions (a) and (b) below, do the following:

- Write out as precise a description as possible using the English language.

- Write a formula in first order logic that captures the required property. You may assume the following:
  - The domain of interpretation is the set of natural numbers \( \mathbb{N} \).
  - You have (unary) function symbols \( a \) and \( b \) to describe the input and output arrays respectively.
  - You have a parameter \( n \) that describes the final index of the input array (so the size of the input array is \( n+1 \), with indices from 0 to \( n \)).
  - You have a binary ‘less than or equal’ predicate \( L \) so that the interpretation of \( L(x,y) \) is \( Ix \leq Iy \), where \( I \) is the interpretation function.
  - You have a binary equality predicate \( E \) so that the interpretation of \( E(x,y) \) is \( Ix = Iy \), where again \( I \) is the interpretation function.

(a) The output array \( b \) is sorted.

(b) The array \( b \) contains exactly the numbers from array \( a \). You may assume that all the entries in \( a \) are different. *Hint: Split the property into two parts.*

(c) Find an example where your previous condition is not sufficient if the array contains duplicate numbers.

(d) Can you see how to improve your previous so that it also works for arrays with duplicate entries? Can you transfer your condition to first order predicate logic? If not, could this be dealt with by expanding the system?

**Exercise 182.** Assume you are given an array of natural numbers. You are tasked with writing a program that finds the median of all these
numbers, that is the number you would find at the halfway point of the array if it were sorted. You may think of this as the ‘middle number’ of all those present.\(^3\)

(a) Write down the properties required to ensure that the number your program finds is indeed the desired median. You may assume that all the numbers in your array are different. \textit{Hint: You may want to use the function }\(|\cdot|\text{ that maps a finite set to the number of elements of that set.}

(b) Now amend your properties so that it can also cope with the case where the same number may appear more than once in the array. \textit{Hint: Consider the set you used in the previous part. Can you think of changing that to a different set that will give you the desired property?}

\begin{exercise}
\textbf{Exercise 183.} Assume you are tasked with writing a program that constructs an individual timetable for each student (better than the one on the Student System) and which therefore needs to know about all the undergraduate students within the School. This means you will require some way of representing each student in your program.

(a) You have to define a function whose source is the set of all undergraduate students in the school, and whose target is something that your programme knows about. What would a suitable target for such a function be?

(b) Which properties should your function have?

(c) Define the function that you would use.

(d) Justify your choice of function by arguing that it satisfies the properties stipulated in Part (b).
\end{exercise}

When when we have a recursively defined procedure we can create a \textit{recurrence relation} (compare Examples 6.46, 6.47 and Exercise 138) that describes its behaviour.

\begin{example}
\textbf{Example 8.1.} Consider the function that sums all the elements from a list of numbers, given in Example 6.11.

\textbf{Base case} \textit{sum.} \hspace{1em} \text{\textit{sum \text{[\text{\text{\}]}\text{ = 0.}}}}^4

\textbf{Step case} \textit{sum.} \hspace{1em} \text{\textit{sum(s : l) = s + sum(l).}}

We want to count the number of additions the function has to carry out for a list with \(n\) elements, which defines a function

\[ f : \mathbb{N} \to \mathbb{N}. \]

We can read off that

\[ f0 = 0, \]
\end{example}

\footnote{The median income of all the employees of a company, for example, is the one with the property that half the employees earn as most as much, and half the employees earn at least as much.}
since in the base case, where the list has 0 elements, no additions are required. The step case shows us that

\[ f(n + 1) = 1 + fn. \]

Hence we have found a recurrence relation as studied in Section 6.4.5. Using the methods from that chapter we can show that the function we have defined is the assignment

\[ n \mapsto n, \]

that is, our function is the identity function on \( \mathbb{N} \).

**Exercise 184.** For the following algorithms, create recursive function definitions similar to those from Examples 6.46, 6.47 and Exercise 138. Is it possible to solve them in the same way as those examples? If not, why not?

(a) The function that calculates the factorial of a number from Example 6.40. Count the number of multiplications the function has to carry out for a given input.

(b) The function that reverses a list from Example 6.12. For \( n \) use the number of elements in the list and count the number of calls the function makes to itself.

(c) The binary search algorithm from Example 4.96. Set \( n \) to be the size of the array and count the number of array look-ups required.

---

4If you were wondering how to deal with the empty list then note that adding no elements at all is usually taken to describe the number 0. More generally, for an operation with unit \( e \), applying the operation to 0 many elements should return \( e \).
Glossary

$\sigma$-algebra
The set of events of a probability space. Contains the whole set of outcomes and is closed under the complement operation and forming unions of countable collections of sets.

absolute, $|\cdot|$
Defined for various sets of numbers, here extended to complex numbers. Given a complex number $a + ib$ we have $|a + ib| = \sqrt{a^2 + b^2}$.

and
Connects two properties or statements, both of which are expected to hold.

anti-symmetric
A binary relation $R$ on a set $S$ is anti-symmetric if $(s, s')$ and $(s', s)$ both being in the relation implies $s = s'$.

argument
The argument of a complex number is the angle it encloses with the positive branch of the real axis.

associative
A binary operation is associative if and only if it gives the same result when applied two three inputs, no matter whether it is first applied to the first two, or first applied to the last two of these.

Bayes’s Theorem
The equality which says that, given events $A$ and $B$, the probability that $B$ given $A$ is the probability that $A$ given $B$, multiplied by the probability of $B$ and divided by that of $A$.

bijective
A function is bijective if and only if it is both, injective and surjective. A bijective function is called a bijection.

binary operation
A function of the type $S \times S \rightarrow S$, which takes two elements of a set $S$ as input and produces another element of $S$.

\[5\]Note that page numbers for Chapters 1–4 will not match the printed notes for these chapters.
**binary relation**

A connection between a source set $S$ and a target set $T$ which is not necessarily a function. It is specified by the collection of all pairs of the form $(s, t)$ in $S \times T$ that belong to it.

**binary tree with labels from a set $S$**

This is a tree where each node has a label from $S$ and where each node has either 0, 1 or 2 children. Formally this is another recursively defined notion.

$\mathbb{C}$

The complex numbers as a set with a number of operations.

**coefficient**

The coefficients of a polynomial are the numbers that appear as factors in front of a power of the variable.

**commutative**

A binary operation is commutative if and only if it gives the same result when its two inputs are swapped.

**complement**

The complement of a set $S$ is always taken with respect to an underlying set, and it consists of those elements of the underlying set which do not belong to $S$.

**composite of functions**

The composite of two functions is defined provided the target of the first is the source of the second. It is the function resulting from taking an element of the source of the first function, applying the first function, and then applying the second to the result.

**composite of partial functions**

Similar to the composite of two functions, but the result is undefined if either of the two functions is not defined where required.

**conditional probability**

Given two events $A$ and $B$, where $B$ has non-zero probability, the conditional probability of $A$ given $B$ is the probability of $A \cap B$ divided by the probability of $B$.

**conjugate, $\overline{z}$**

The conjugate $\overline{z}$ of a complex number number $z = a + ib$ is $a - ib$.

**continuous**

A random variable is continuous if and only if it is not discrete.

**countable**

A set is countable if and only if there is an injective function from it to $\mathbb{N}$.
countably infinite
A set is countably infinite if it is both, countable and infinite.

cumulative distribution function (cdf)
The cdf of a random variable maps each element \( r \) of \( \mathbb{R} \) to the probability that the random variable has a value less than or equal to \( r \).

definition by cases
A way by piecing together functions to give a new function.

degree of a polynomial
The degree of a polynomial is the largest index whose coefficient is unequal to 0.

directed graph
A set (of nodes) connected by edges; can be described using a binary relation on the set.

discrete
A random variable is discrete if and only if its range is countable.

disjoint
Two sets are disjoint if they have no elements in common.

div
The (integer) quotient of two numbers when using integer division.

divides
A number \( m \) divides a number \( n \) in some set of numbers if there exists a number \( k \) with the property that \( n = km \).

divisible
We say for natural numbers (or integers) that \( n \) is divisible by \( m \) if and only if \( n \) leaves remainder 0 when divided by \( m \) using integer division.

domain of definition
For a partial function it is the set consisting of all those elements of the source set for which the partial function is defined.

dominate
A function \( f \) from a set \( X \) to \( \mathbb{N}, \mathbb{Z}, \mathbb{Q} \) or \( \mathbb{R} \) dominates another \( g \) with the same source and target if and only if the graph of \( f \) lies entirely above the graph of \( g \) (graphs touching is allowed).

empty set, \( \emptyset \)
A set which has no elements.

equivalence class with respect to \( R \) generated by \( s \), \([s]\)
The set of all elements which are related to \( s \) by the equivalence relation \( R \).
equivalence relation
A binary relation on a set is an equivalence relation if it is reflexive, symmetric and transitive.

equivalence relation generated by a binary relation $R$
The transitive closure of the symmetric closure of the reflexive closure of $R$.
even
An integer (or natural number) is even if and only if it is divisible by 2.
eventually dominate
A function $f$ from $\mathbb{N}$ to $\mathbb{N}$ eventually dominates another $g$ with the same source and target if and only if there is some number beyond which the graph of $f$ lies above that of $g$ (graphs touching is allowed). The analogous definition works for functions with source and target $\mathbb{Z}$, $\mathbb{Q}$ or $\mathbb{R}$.

expected value
The expected value of a random variable can be thought of as the average value it takes. It is given by the integral of the product of a number which the probability that it is the value of the random variable. If the random variable is discrete then this is given by a sum.

for all
Expresses a statement or property that holds for all the entities specified.

full binary tree with labels from a set $S$
This is a tree where each node has a label from $S$ and where each node has either 0 or 2 children. Formally this is another recursively defined notion.

function
A function has a source and a target, and contains instructions to turn an element of the source set into an element of the target set. Where partial functions are discussed sometimes known as total function.

graph of a function
The graph of a function $f$ with source $S$ and target $T$ consists of all those pairs in $S \times T$ which are of the form $(s, fs)$.

greatest element, $\top$
An element which is greater than or equal to every element of the given poset.

greatest lower bound, infimum
An element of a poset $(P, \leq)$ is a greatest lower bound of a given subset of that poset if it is both, a lower bound and greater than or equal to every lower bound of the given set.
group

A set with an associative binary operation which has a unit and in which every element has an inverse.

identity function

The identity function on a set is a function from that set to itself which returns its input as the output.

identity relation

The identity relation on a set $S$ relates every element of $S$ to itself, and to nothing else.

if and only if

Connects two properties or statement, and it is expected that one holds precisely when the other holds.

image of a set, $f[\cdot]$

The image of a set consists of the images of all its elements, and one writes $f[S]$ for the image of the set $S$ under the function $f$.

image of an element

The image of an element under a function is the output of that function for the given element as the input.

imaginary part

Every complex number $a + bi$ has an imaginary part $b$.

implies

Connects two properties or statements, and if the first of these holds then the second is expected to also hold.

independent

Two events are independent if and only if the probability of their intersection is the product of their probabilities. Two random variables are independent if and only if for every two events it is the case that the probability that the two variables take values in the product of those events is the product of the probabilities that each random variable takes its value in the corresponding event..

infinite

A set is infinite if and only if there is an injection from it to a proper subset.

injective

A function is injective if and only if the same output can only arise from having the same input. An injective function is called an injection.

integer

A whole number that may be positive or negative.
integer division

Integer division is an operation on integers; given two integers \( n \) and \( m \) where \( m \neq 0 \), we get an integer quotient \( n \div m \) and a remainder \( n \mod m \).

intersection, \( \cap \)

The intersection of two sets \( S \) and \( T \) is written as \( S \cap T \), and it consists of all the elements of the underlying set that belong to both, \( S \) and \( T \). The symbol \( \cap \) is used for the intersection of a collection number of sets.

inverse

One element is the inverse for another with respect to a binary operation if and only if when using the two elements as inputs (in either order) to the operation the output is the unit.

inverse function

A function is the inverse of another if and only if the compose (either way round) to give an identity function.

law of total probability

A rule that allows us to express the probability of an event from probabilities that split the event up into disjoint parts.

least element, \( \bot \)

An element which is less than or equal to every element of the given poset.

least upper bound, supremum

An element of a poset \((P, \leq)\) is a least upper bound of a given subset of that poset if it is both, a upper bound and less than or equal to every upper bound of the given set.

list over a set \( S \)

A list over a set \( S \) is a recursively defined concept consisting of an ordered tuple of elements of the given set.

lower bound

An element of a poset \((P, \leq)\) is a lower bound for a given subset of \( P \) if it is less than or equal to every element of that set.

maximal element

An element which does not have any elements above it.

measurable

A function from the sample set of a probability space to the real numbers is measurable if and only if for every interval it is the case that the set of all outcomes mapped to that interval is an event.
**minimal element**

An element which does not have any elements below it.

**mod**

The remainder when using integer division.

**monoid**

A set with an associative binary operation which has a unit.

**multiplication law**

The equality which says that given events $A$ and $B$, the probability of the intersection of $A$ and $B$ is that of $A$ given $B$ multiplied with that of $B$.

**N**

The natural numbers as a set with a number of operations. This set and its operations are formally defined in Section 6.4.

**natural number**

One of the ‘counting numbers’, 0, 1, 2, 3, . . .

**odd**

An integer (or natural number) number is odd if it is not even or, equivalently, if it leaves a remainder of 1 when divided by 2.

**opposite relation of $R$, $R^{op}$**

The relation consisting of those pairs $(t, s)$ for which $(s, t)$ is in $R$.

**or**

Connects two properties or statements, at least one of which is expected to hold.

**ordered binary tree with labels from a set $S$**

Such a tree is ordered if the set $S$ is ordered, and if for every node, all the nodes in the left subtree have a label below that of the current node, while all the nodes in the right subtree have a label above.

**pairwise disjoint**

A collection of sets has this property if any two of them have an empty intersection.

**partial order**

A binary relation on a set is a partial order provided it is reflexive, anti-symmetric and transitive.

**polynomial equation**

An equation of the form $\sum_{i=0}^{n} a_i x^i$. 

**450**
polynomial function
A function from numbers to numbers whose instruction is of the form $x$ is mapped to $\sum_{i=1}^{n} a_i x^i$ (where the $a_i$ are from the appropriate set of numbers).

poset
A set with a partial order, also known as a partially ordered set.

powerset, $\mathcal{P}$
The powerset of a set $S$ is the set of all subsets of $S$.

prime
A natural number or an integer is prime if its dividing a product implies its dividing one of the factors.

probability density function
A function from some real interval to $\mathbb{R}^+$ with the property that its integral over the interval is 1 and whose integral over subintervals always exists.

probability distribution
A function from the set of events that has the property that the probability of a countable family of pairwise disjoint sets is the sum of the probabilities of its elements.

probability mass function (pmf)
The pmf of a discrete random variable maps each element of the range of that random variable to the probability that it occurs.

probability space
A sample set together with a set of events and a probability distribution.

product of two sets
A way of forming a new set by taking all the ordered pairs whose first element is from the first set, and whose second element is from the second set.

proper subset
A set $S$ is a proper subset of the set $T$ if and only if $S$ is a subset of $T$ and there is at least one element of $T$ which is not in $S$.

$\mathbb{Q}$
The set of all rational numbers together with a variety of operations, formally defined in Definition 0.1.3.

$\mathbb{R}^+$
The set of all real numbers greater than or equal to 0.
\( \mathbb{R} \)

The set of all real numbers.

**random variable**

A random variable is a measurable function from the set of outcomes of some probability space to the real numbers.

**range of a function**

The range of a function is the set of all elements which appear as the output for at least one of the inputs, that is, it is the collection of the images of all the possible inputs.

**rational number**

A number is rational if it can be written as the fraction of two integers. A formal definition is given on page 406 (and the preceding pages).

**real number**

We do not give a formal definition of the real numbers in this text.

**real part**

Every complex number \( a + bi \) has a real part \( a \).

**reflexive**

A binary relation on a set is reflexive if it relates each element of the set to itself.

**reflexive closure**

The reflexive closure of a binary relation on a set \( S \) is formed by adding all pairs of the form \((s, s)\) to the relation.

**relational composite**

A generalization of composition for (partial) functions.

**remainder for integer division**, \( \text{mod} \)

The integer \( n \mod m \) is defined to be the remainder left when dividing \( n \) by \( m \) in the integers.

**set difference**, \( \setminus \)

The set difference \( S \setminus T \) consists of all those elements of \( S \) which are not in \( T \).

**size of a set**

A set is smaller than another if there exists an injective function from the first to the second. They have the same size if they are both smaller than the other.

**standard deviation**

The standard deviation of a random variable is given by the square root of its variance.
string over a set \( S \)

A formal word constructed by putting together symbols from \( S \).

**surjective**

A function is surjective if and only if every element of the target appears as the output for at least one element of the input. This means that the image of the function is the whole target set. A surjective function is called a surjection.

**symmetric closure**

The symmetric closure of a binary relation on a set is formed by taking the union of the relation with its opposite.

**there exists**

Expresses the fact that a statement or property holds for at least one of the entities specified.

**total order**

A total order is a partial order in which every two elements are comparable.

**transitive**

A binary relation on a set \( S \) is transitive provided that \((s, s')\) and \((s', s'')\) being in the relation implies that \((s, s'')\) is in the relation.

**transitive closure**

The transitive closure of a binary relation on a set is formed by adding all pairs of elements \((s_1, s_n)\) for which there is a list of elements \(s_1, s_2\) to \(s_n\) in \( S \) such that \((s_i, s_{i+1})\) is in the relation.

**union, \( \cup \)**

The union of two sets \( S \) and \( T \) is written as \( S \cup T \). It consists of all elements of the underlying set that belong to at least one of \( S \) and \( T \). The symbol \( \bigcup \) is used for the union of a collection of sets.

**uncountable**

A set is uncountable if it is not countable.

**unique existence**

A more complicated statement requiring the existence of an entity, and the fact that this entities is unique with the properties specified.

**unit**

An element of a set is a unit for a binary operation on that set if and only if applying the operation to that, plus any of the other elements, returns that other element.

**upper bound**

An element of a poset \((P, \leq)\) is an upper bound for a given subset of \( P \) if it is greater than or equal to every element of that set.
The variance of a random variable with expected value $e$ is given by the expected value of the random variable constructed by squaring the result of subtracting $e$ from the original random variable.

The integers with various operations, see Definition 0.1.2 for a formal account.
Core Exercises marked this week

Exercise 105 on page 268. Do any part.

Exercise 110 on page 295.

Exercise 109 on page 294.

Extensional Exercises marked this week

Exercise 101 on page 264. Do any one part.

Exercise 112 on page 296. Do any three parts.

Remember that

• the deadline is the beginning of the examples class, and that you have to be able to promptly answer questions by the TA, referring to your rough work as needed;

• you may only use concepts which are defined in these notes (Chapter 0 establishes concepts for numbers), and for every concept you do use you should find the definition in the notes and work with that;

• you should justify each step in your proofs;

• if you are stuck on an exercise move on to the next one after ten minutes, but write down why you got stuck so that you can explain that to the TA in the examples class.

• In order for you to be considered as having seriously tried an exercise you must have written down the number of relevant examples and definitions in the notes, and you must be able to tell the TA what you tried to apply those ideas to the given problem. If you didn’t understand the examples/definitions you must be able to tell them where exactly your understanding failed.

You should now be ready to do all the exercises in Chapter 5 and Section 6.1.
COMP11120, Semester 2
Exercise Sheet 12
For examples classes in Week 3

Core Exercises marked this week

Exercise 115 on page 300. Do any two parts.

Exercise 116 on page 309.

Exercise 118 on page 310. Do (a) and (b) and any of the remaining parts.

Extensional Exercises marked this week

Exercise 119 on page 310.

Exercise 123 on page 312.

Remember that

- the deadline is the beginning of the examples class, and that you have to be able to promptly answer questions by the TA, referring to your rough work as needed;

- you may only use concepts which are defined in these notes (Chapter 0 establishes concepts for numbers), and for every concept you do use you should find the definition in the notes and work with that;

- you should justify each step in your proofs;

- if you are stuck on an exercise move on to the next one after ten minutes, but write down why you got stuck so that you can explain that to the TA in the examples class.

- In order for you to be considered as having seriously tried an exercise you must have written down the number of relevant examples and definitions in the notes, and you must be able to tell the TA what you tried to apply those ideas to the given problem. If you didn’t understand the examples/definitions you must be able to tell them where exactly your understanding failed.

You should be able to do all exercises in Section 6.2.
COMP11120, Semester 2
Exercise Sheet 13
For examples classes in Week 4

Core Exercises marked this week

Exercise 125 on page 315.

Exercise 128 on page 328.

Exercise 130 on page 329. Do part (a) and one of (c) or (d).

Extensional Exercises marked this week

Exercise 126 on page 315.

Exercise 133 on page 331. Do one part from (a)–(c) and one from (d)–(f).

Remember that

- the **deadline** is the beginning of the examples class, and that you have to be able to promptly answer questions by the TA, referring to your rough work as needed;

- you may only use concepts which are defined in these notes (Chapter 0 establishes concepts for numbers), and for every concept you do use you should find the definition in the notes and work with that;

- you should justify each step in your proofs;

- if you are stuck on an exercise move on to the next one after ten minutes, but write down why you got stuck so that you can explain that to the TA in the examples class.

- In order for you to be considered as having seriously tried an exercise you must have written down the number of relevant examples and definitions in the notes, and you must be able to tell the TA what you tried to apply those ideas to the given problem. If you didn’t understand the examples/definitions you must be able to tell them where exactly your understanding failed.

You should be able to do all exercises up to and including Section 6.4.2 now.
Core Exercises marked this week

Exercise 136 on page 337. Do any two parts.

Exercise 138 on page 340. Do one part from (a)–(c) and one from (d)–(g).

Exercise 142 on page 356.

Extensional Exercises marked this week

Exercise 139 on page 343. Do any part.

Exercise 143 on page 356.

Remember that

- the deadline is the beginning of the examples class, and that you have to be able to promptly answer questions by the TA, referring to your rough work as needed;

- you may only use concepts which are defined in these notes (Chapter 0 establishes concepts for numbers), and for every concept you do use you should find the definition in the notes and work with that;

- you should justify each step in your proofs;

- if you are stuck on an exercise move on to the next one after ten minutes, but write down why you got stuck so that you can explain that to the TA in the examples class.

- In order for you to be considered as having seriously tried an exercise you must have written down the number of relevant examples and definitions in the notes, and you must be able to tell the TA what you tried to apply those ideas to the given problem. If you didn’t understand the examples/definitions you must be able to tell them where exactly your understanding failed.

You should now be able to do Exercises up to Section 7.2.
COMP11120, Semester 2
Exercise Sheet 15
For examples classes in Week 6

Core Exercises marked this week

Exercise 149 on page 373. Do one part from (a)–(b), one from (c)–(e) and one from (f)–(h).

Exercise 151 on page 376. Do one part from (a)–(b), one from (c)–(e), one from (f)–(h) and one from (i)–(k).

Exercise 157\(^6\) on page 391. Do any one part.

Extensional Exercises marked this week

Exercise 150 on page 374.

Exercise 158 on page 400. Do all parts; it is sufficient to carry out two calculations each time instead of all four.

Remember that

- the deadline is the beginning of the examples class, and that you have to be able to promptly answer questions by the TA, referring to your rough work as needed;

- you may only use concepts which are defined in these notes (Chapter 0 establishes concepts for numbers), and for every concept you do use you should find the definition in the notes and work with that;

- you should justify each step in your proofs;

- if you are stuck on an exercise move on to the next one after ten minutes, but write down why you got stuck so that you can explain that to the TA in the examples class.

- In order for you to be considered as having seriously tried an exercise you must have written down the number of relevant examples and definitions in the notes, and you must be able to tell the TA what you tried to apply those ideas to the given problem. If you didn’t understand the examples/definitions you must be able to tell them where exactly your understanding failed.

You should now be ready to do all exercises up to and including Section 7.3.5.

\(^6\)Two exercises on this sheet are out of sequence to see whether this helps with better distributing load. Last year students felt that Sheet 16 contains too many different new things.
Core Exercises marked this week

Exercise 153\(^7\) on page 382. Do one from (a)–(c), one from (d)–(f) and one from (g)–(h).

Exercise 154 on page 385.

Exercise 169 on page 417. Do one part from (a)–(d), one from (e)–(h), one from (i)–(k) and one from (l)–(o).

Extensional Exercises marked this week

Exercise 165 on page 410.

Exercise 164 on page 409.

Remember that

- the **deadline** is the beginning of the examples class, and that you have to be able to promptly answer questions by the TA, referring to your rough work as needed;

- you may only use concepts which are defined in these notes (Chapter 0 establishes concepts for numbers), and for every concept you do use you should find the definition in the notes and work with that;

- you should justify each step in your proofs;

- if you are stuck on an exercise move on to the next one after ten minutes, but write down why you got stuck so that you can explain that to the TA in the examples class.

- In order for you to be considered as having seriously tried an exercise you must have written down the number of relevant examples and definitions in the notes, and you must be able to tell the TA what you tried to apply those ideas to the given problem. If you didn’t understand the examples/definitions you must be able to tell them where exactly your understanding failed.

You should now be ready to do all exercises up to and including Section 7.4.1.

---

\(^7\) The first two exercises on this sheet were originally on Sheet 15. The change is an experiment to better cope with a high load of new ideas on this sheet.
Core Exercises marked this week

Exercise 174 on page 428. Do one part from (a)–(c), one from (d)–(f), one from (g)–(j) and one from (k)–(l).

Exercise 178 on page 435. Do one part from (a)–(c) and one from (d)–(e).

Exercise 181 on page 440.

Extensional Exercises marked this week

Exercise 175 on page 429. Do all parts, but in each part you only have to do one of the statements (for greatest/least, or maximal/minimal).

Exercise 180 on page 436.

Remember that

- the deadline is the beginning of the examples class, and that you have to be able to promptly answer questions by the TA, referring to your rough work as needed;

- you may only use concepts which are defined in these notes (Chapter 0 establishes concepts for numbers), and for every concept you do use you should find the definition in the notes and work with that;

- you should justify each step in your proofs;

- if you are stuck on an exercise move on to the next one after ten minutes, but write down why you got stuck so that you can explain that to the TA in the examples class.

- In order for you to be considered as having seriously tried an exercise you must have written down the number of relevant examples and definitions in the notes, and you must be able to tell the TA what you tried to apply those ideas to the given problem. If you didn’t understand the examples/definitions you must be able to tell them where exactly your understanding failed.

You should now be ready to do all exercises in the notes.