Theoretical Computer Science I: Logic

— Winter 2018 —

DHBW Mannheim

Prof. Dr. Karl Stroetmann

13. November 2018

These lecture notes, the corresponding LaTeX sources and the programs discussed in these lecture notes are available at

https://github.com/karlstroetmann/Logik.

The lecture notes can be found in the dictionary Lecture-Notes-Python in the file logic.pdf. As I am currently switching from using the programming language SetlX to using *Python* instead, these lecture notes are being constantly revised. At the moment, the lecture notes still contain SetlX programs. My goal is to replace these programs with equivalent *Python* programs. In order to automatically update the lecture notes, you can install the program git. Then, using the command line, you can clone my repository using the command

```
git clone https://github.com/karlstroetmann/Logik.git.
```

Once the repository has been cloned, it can be updated using the command

```
git pull.
```

Inhaltsverzeichnis

Kapitel 1

Introduction

In this short chapter, I would like to motivate why it is that you have to learn logic when you study computer science. After that, I will give a short overview of the lecture.

1.1 Motivation

Modern software systems are among the most complex systems developed by mankind. You can get a sense of the complexity of these systems if you look at the amount of work that is necessary to build and maintain complex software systems. For example, in the telecommunication industry it is quite common that software projects require more than a thousand collaborating developers to develop a new system. Obviously, the failure of a project of this size is very costly. The page

Staggering Impact of IT Systems Gone Wrong

presents a number of examples showing big software projects that have failed and have subsequently caused huge financial losses. These examples show that the development of complex software systems requires a high level of precision and diligence. Hence, the development of software needs a solid scientific foundation. Both mathematical logic and set theory are important parts of this foundation. Furthermore, both set theory and logic have immediate applications in computer science.

1. Logic can be used to specify the interfaces of complex systems.
2. The correctness of digital circuits can be verified using automatic theorem provers that are based on propositional logic.
3. Set theory and the theory of relations is one of the foundations of relational databases.

It is easy to extend this enumeration. However, besides their immediate applications, there is another reason you have to study both logic and set theory: Without the proper use of abstractions, complex software systems cannot be managed. After all, nobody is able to keep millions of lines of program code in her head. The only way to construct and manage a software system of this size is to introduce the right abstractions and to develop the system in layers. Hence, the ability to work with abstract concepts is one of the main virtues of a modern computer scientist. Exposing students to logic and set theory trains their abilities to work with abstract concepts.

From my past teaching experience I know that many students think that a good programmer already is a good computer scientist. However, a good programmer need not be a scientist, while a computer scientist, by its very name, is a scientist. There is no denying that mathematics in general and logic in particular is an important part of science, so you should master it. Furthermore, this part of your education is much more permanent than the knowledge of a particular programming language. Nobody knows which programming language will be *en vogue* in 10 years from now. In three years, when you start your professional career, quite

a lot of you will have to learn a new programming language. What will count then will be much more your ability to quickly grasp new concepts rather than your skills in a particular programming language.

1.2 Overview

The first lecture in theoretical computer science creates the foundation that is needed for future lectures. This lecture deals mostly with mathematical logic and is structured as follows.

1. We begin our lecture with a short introduction of set theory. A basic understanding of set theory is necessary for us to formally define the semantics of both propositional logic and first order logic.

2. We proceed to introduce the programming language *Python*.

   As the concepts introduced in this lecture are quite abstract, it is beneficial to clarify the main ideas presented in this lectures via programs. The programming language *Python* supports both sets and their operations and is therefore suitable to implement most of the abstract ideas presented in this lecture. According to the IEEE (Institute of Electrical and Electronics Egineers), *Python* is now the most popular programming language. Furthermore, *Python* is now the most popular introductory teaching language at top U.S. universities. For these reasons I have decided to base these lectures on *Python*.

3. Next, we investigate the limits of computability.

   For certain problems there is no algorithm that can solve the problem algorithmically. For example, the question whether a given program will terminate for a given input is not decidable. This is known as the halting problem. We will prove the undecidability of the halting problem in the third chapter.

4. The fourth chapter discusses propositional logic.

   In logic, we distinguish between propositional logic, first order logic, and higher order logic. Propositional logic is only concerned with the logical connectives

   $$\text{“}\neg\text{”}, \text{“}\wedge\text{”}, \text{“}\vee\text{”}, \text{“}\rightarrow\text{”} \text{ und } \text{“}\leftrightarrow\text{”},$$

   while first-order logic also investigates the quantifiers

   $$\text{“}\forall\text{” and “}\exists\text{”},$$

   where these quantifiers range over the objects of the domain of discourse. Finally, in higher order logic the quantifiers also range over functions and predicates.

   As propositional logic is easier to grasp than first-order logic, we start our investigation of logic with propositional logic. Furthermore, propositional logic has the advantage of being decidable: We will present an algorithm that can check whether a propositional formula is universally valid. In contrast to propositional logic, first-order logic is not decidable.

   Next, we discuss applications of propositional logic: We will show how the 8 queens problem can be reduced to the question, whether a formula from propositional logic is satisfiable. We present the algorithm of Davis and Putnam that can decide the satisfiability of a propositional formula. This algorithm is therefore able to solve the 8 queens problem.

5. Finally, we discuss first-order logic.

   The most important concept of the last chapter will be the notion of a formal proof in first order logic. To this end, we introduce a formal proof system that is complete for first order logic. Completeness means that we will develop an algorithm that can prove the correctness of every first-order formula that is universally valid. This algorithm is the foundation of automated theorem proving.

   As an application of theorem proving we discuss the systems Prover9 and Mace4. Prover9 is an automated theorem prover, while Mace4 can be used to refute a mathematical conjecture.

Kapitel 2

Naive Set Theory

The concept of set theory has arisen towards the end of the 19th century from an effort to put mathematics on a solid foundation. The creation of a solid foundation was considered necessary as the concept of infinity increasingly worried mathematicians.

The essential parts of set theory have been defined by Georg Cantor (1845 – 1918). The first definition of the concept of a set was approximately as follows [Can95]:

> A "set" is a well-defined collection M of certain objects x of our perception or our thinking.

Here, the attribute "well-defined" expresses the fact that for a given quantity M and an object x we have to be able to decide whether the object x belongs to the set M. If x belongs to M, then x is called an element of the set M and we write this as

$$x \in M.$$

The symbol "$\in$" is therefore used in set theory as a binary predicate symbol. We use infix notation when using this symbol, that is we write $x \in M$ instead of $\in(x, M)$. Slightly abbreviated we can define the notion of a set as follows:

A set is a well-defined collection of elements.

To mathematically understand the concept of a well-defined collection of elements, Cantor introduced the so-called axiom of comprehension. We can formalize this axiom as follows: If $p(x)$ a property that an object x can have, we can define the set M of all objects that have this property. Therefore, the set M can be defined as

$$M := \{x \mid p(x)\}$$

and we read this definition as "M is the set of all x such that $p(x)$ holds". Here, a property $p(x)$ is just a formula in which the variable x happens to appear. We illustrate the axiom of comprehension by an example: If $\mathbb{N}$ is the set of natural numbers, then we can define the set of all even numbers via the property

$$p(x) \ := \ (\exists y \in \mathbb{N} : x = 2 \cdot y).$$

Using this property, the set of even numbers can be defined as

$$\{x \mid \exists y \in \mathbb{N} : x = 2 \cdot y\}.$$

Unfortunately, the unrestricted use of the axiom of comprehension leads to serious problems. To give an example, let us consider the property of a set to not contain itself. Therefore, we define

$$p(x) := \neg(x \in x)$$

and further define the set R as follows:

$$R := \{x \mid \neg(x \in x)\}.$$

Intuitively, we might expect that no set can contain itself. However, things turn out to be more complicated. Let us try to check whether the set R contains itself. We have

$$\begin{aligned} & R \in R \\ \Leftrightarrow \quad & R \in \{x \mid \neg(x \in x)\} \\ \Leftrightarrow \quad & \neg(R \in R). \end{aligned}$$

So we have shown that

$$R \in R \;\Leftrightarrow\; \neg(R \in R)$$

holds. Obviously, this is a contradiction. As a way out, we can only conclude that the expression

$$\{x \mid \neg(x \in x)\}$$

does not define a set. This shows that the axiom of comprehension is too general: Not every expression of the form

$$M := \{x \mid p(x)\}$$

defines a set. The expression

$$\{x \mid \neg(x \in x)\}$$

has been found by the British logician and philosopher Bertrand Russell (1872 – 1970). It is known as Russell's Antinomy.

In order to avoid paradoxes such as Russell's antinomy, it is necessary to be more careful when sets are constructed. In the following, we will present methods to construct sets that are weaker than the axiom of comprehension, but, nevertheless, these methods will be sufficient for our purposes. We will use the notation underlying the comprehension axiom and write set definitions in the form

$$M = \{x \mid p(x)\}.$$

However, we won't be allowed to use arbitrary formulas $p(x)$ here. Instead, the formulas we are going to use for $p(x)$ have to satisfy some restrictions. These restrictions will prevent the construction of self-contradictory sets.

2.1 Defining Sets by Listing their Elements

The simplest way to define a set is to list of all of its elements. These elements are enclosed in the curly braces "{" and "}" and are separated by commas. For example, when we define

$$M := \{1, 2, 3\},$$

then the set M contains the elements 1, 2 and 3. Using the notation of the axiom of comprehension we could write this set as

$$M = \{x \mid x = 1 \vee x = 2 \vee x = 3\}.$$

Another example of a set that can be created by explicitly enumerating its elements is the set of all lower case Latin letters. This set is given as define:

$$\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}.$$

Occasionally, we will use dot notation to define a set. Using dot notation, the set of all lower case elements is written as

$$\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \cdots, \mathtt{x}, \mathtt{y}, \mathtt{z}\}.$$

Of course, if we use dot notation the interpretation of the dots "$\cdots$" must always be obvious from the context of the definition.

As a last example, we consider the empty set $\emptyset$, which is defined as

$$\emptyset := \{\}.$$

Therefore, the empty set does not contain any element at all. This set plays an important role in set theory. This role is similar to the role played by the number 0 in algebra.

If a set is defined by listing all of its elements, the order in which the elements are listed is not important. For example, we have

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

since both sets contain the same elements.

2.2 Predefined Infinite Sets of Numbers

All sets that are defined by explicitly listing their elements can only have finitely many elements. In mathematics there are a number of sets that have an infinite number of elements. One example is the set of natural numbers, which is usually denoted by the symbol $\mathbb{N}$. Unlike some other authors, I regard the number zero as a natural number. This is consistent with the ISO-standard 31-11.[1] Given the concepts discussed so far, the quantity $\mathbb{N}$ cannot be defined. We must therefore demand the existence of this set as an axiom. More precisely, we postulate that there is a set $\mathbb{N}$ which has the following three properties:

1. $0 \in \mathbb{N}$.
2. If we have a number n such that $n \in \mathbb{N}$, then we also have $n + 1 \in \mathbb{N}$.
3. The set $\mathbb{N}$ is the smallest set satisfying the first two conditions.

We write

$$\mathbb{N} := \{0,1,2,3,\cdots\}.$$

Along with the set $\mathbb{N}$ of natural numbers we will use the following sets of numbers:

1. $\mathbb{N}^*$ is the set of positive natural numbers, so we have
   $$\mathbb{N}^* := \{n \mid n \in \mathbb{N} \wedge n > 0\}.$$
2. $\mathbb{Z}$ is the set of integers, we have
   $$\mathbb{Z} = \{0,1,-1,2,-2,3,-3,\cdots\}$$
3. $\mathbb{Q}$ is the set of rational numbers, we have
   $$\left\{\frac{p}{q} \;\middle|\; p \in \mathbb{Z} \wedge q \in \mathbb{N}^*\right\}.$$
4. $\mathbb{R}$ is the set of real numbers.

   A clean mathematically definition of the notion of a real number requires a lot of effort and is out of the scope of this lecture. If you are interested, a detailed description of the construction of real numbers is given in my lecture notes on Analysis.

2.3 The Axiom of Specification

The axiom of specification, also known as the axiom of restricted comprehension, is a weakening of the comprehension axiom. The idea behind the axiom of specification is to use a property p to select from an existing set M a subset N of those elements that have the property $p(x)$:

[1] The ISO standard 31-11 has been replaced by the ISO-standard 80000-2, but the definition of the set $\mathbb{N}$ has not changed. In the text, I did not cite ISO 80000-2 because the content of this standard is not freely available, at least not legally.

$$N := \{x \in M \mid p(x)\}$$

In the notation of the axiom of comprehension this set is written as

$$N := \{x \mid x \in M \wedge p(x)\}.$$

This is a restricted form of the axiom of comprehension, because the condition "$p(x)$" that was used in the axiom of comprehension is now strengthened to the condition "$x \in M \wedge p(x)$".

Example: Using the axiom of restricted comprehension, the set of even numbers can be defined as

$$\{x \in \mathbb{N} \mid \exists y \in \mathbb{N} : x = 2 \cdot y\}.$$

2.4 Power Sets

In order to introduce the notion of a power set we first have to define the notion of a subset. If M and N are sets, then M is a subset of N if and only if each element of the set M is also an element of the set N. In that case, we write $M \subseteq N$. Formally, we define

$$M \subseteq N \stackrel{\text{def}}{\iff} \forall x : (x \in M \rightarrow x \in N).$$

Example: We have

$$\{1,3,5\} \subseteq \{1,2,3,4,5\}.$$

Furthermore, for any set M we have that

$$\emptyset \subseteq M.$$ ◇

The power set of a set M is now defined as the set of all subsets of M. We write 2^M for the power set of M. Therefore we have

$$2^M := \{x \mid x \subseteq M\}.$$

Example: Let us compute the power set of the set $\{1,2,3\}$. We have

$$2^{\{1,2,3\}} = \big\{\{\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{2,3\}, \{1,2,3\}\big\}.$$

This set has $8 = 2^3$ elements. ◇

In general, if the set M has m different elements, then it can be shown that the power set 2^M has 2^m different elements. More formally, let us designate the number of elements of a finite set M as $card(M)$. Then we have

$$card\left(2^M\right) = 2^{card(M)}.$$

This explains the notation 2^M to denote the power set of M.

2.5 The Union of Sets

If two sets M and N are given, the union of M and N is the set of all elements that are either in the set M or in the set N or in both M and in N. This set is written as $M \cup N$. Formally, this set is defined as

$$M \cup N := \{x \mid x \in M \vee x \in N\}.$$

Example: If $M = \{1,2,3\}$ and $N = \{2,5\}$, we have

$$\{1,2,3\} \cup \{2,5\} = \{1,2,3,5\}.$$ ◇

The concept of the union of two sets can be generalized. Consider a set X such that the elements of X are sets themselves. For example, the power set of a set M is a set whose elements are sets themselves. We can form the union of all the sets that are elements of the set X. We write this set as $\bigcup X$. Formally, we have

$$\bigcup X := \{y \mid \exists x \in X : y \in x\}.$$

Example: If we have

$$X = \big\{ \{\}, \{1,2\}, \{1,3,5\}, \{7,4\} \big\},$$

then

$$\bigcup X = \{1,2,3,4,5,7\}.$$ ◇

Exercise 1: Assume that M is a subset of $\mathbb{N}$. Compute the set $\bigcup 2^M$. ◇

2.6 The Intersection of Sets

If two sets M and N are given, we define the intersection of M and N as a set of all objects that are elements of both M and N. We write that set as the average $M \cap N$. Formally, we define

$$M \cap N := \{x \mid x \in M \wedge x \in N\}.$$

Example: We calculate the intersection of the sets $M = \{1,3,5\}$ and $N = \{2,3,5,6\}$. We have

$$M \cap N = \{3,5\}.$$ ◇

The concept of the intersection of two sets can be generalized. Consider a set X such that the elements of X are sets themselves. We can form the intersection of all the sets that are elements of the set X. We write this set as $\bigcap X$. Formally, we have

$$\bigcap X := \{y \mid \forall x \in X : y \in x\}.$$

Exercise 2: Assume that M is a subset of $\mathbb{N}$. Compute the set $\bigcap 2^M$. ◇

2.7 The Difference of Sets

If M and N are sets, we define the difference of M and N as the set of all objects from M that are not elements of N. The difference of the sets M and N is written as $M \backslash N$ and is formally defined as

$$M \backslash N := \{x \mid x \in M \wedge x \notin N\}.$$

Example: We compute the difference of the sets $M = \{1,3,5,7\}$ and $N = \{2,3,5,6\}$. We have

$$M \backslash N = \{1,7\}.$$ ◇

2.8 Image Sets

If M is a set and f is a function defined for all x of M, then the image of M under f is defined as follows:

$$f(M) := \{y \mid \exists x \in M : y = f(x)\}.$$

This set is also written as

$$f(M) := \{f(x) \mid x \in M\}.$$

Example: The set Q of all square numbers can be defined as

$$Q := \{y \mid \exists x \in \mathbb{N} : y = x^2\}.$$

Alternatively, we can define this set as

$$Q := \{x^2 \mid x \in \mathbb{N}\}.$$ ◇

2.9 Cartesian Products

In order to be able to present the notion of a Cartesian product, we first have to introduce the notion of an ordered pair of two objects x and y. The ordered pair of x and y is written as

$$\langle x, y \rangle.$$

In the literature, the ordered pair of x and y is sometimes written as (x, y), but I prefer the notation with angle brackets. The first component of the pair $\langle x, y \rangle$ is x, while y is the second component. Two ordered pairs $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$ are equal if and only if they have the same first and second component, i.e. we have

$$\langle x_1, y_1 \rangle = \langle x_2, y_2 \rangle \;\Leftrightarrow\; x_1 = x_2 \wedge y_1 = y_2.$$

The Cartesian product of two sets M and N is now defined as the set of all ordered pairs such that the first component is an element of M and the second component is an element of N. Formally, we define the cartesian product $M \times N$ of the sets M and N as follows:

$$M \times N := \big\{ z \mid \exists x \colon \exists y \colon \big(z = \langle x, y \rangle \wedge x \in M \wedge y \in N \big) \big\}.$$

To be more concise we usually write this as

$$M \times N := \big\{ \langle x, y \rangle \mid x \in M \wedge y \in N \big\}.$$

Example: If $M = \{1, 2, 3\}$ and $N = \{5, 7\}$ we have

$$M \times N = \big\{ \langle 1, 5 \rangle, \langle 2, 5 \rangle, \langle 3, 5 \rangle, \langle 1, 7 \rangle, \langle 2, 7 \rangle, \langle 3, 7 \rangle \big\}. \qquad \diamond$$

The notion of an ordered pair can be generalized to the notion of an n-tuple where n is a natural number: An n-tuple has the form

$$\langle x_1, x_2, \cdots, x_n \rangle.$$

In a similar way, we can generalize the notion of a Cartesian product of two sets to the Cartesian product of n sets. The general Cartesian product of n sets $M_1, \cdots, M_n$ is defined as follows:

$$M_1 \times \cdots \times M_n = \big\{ \langle x_1, x_2, \cdots, x_n \rangle \;\big|\; x_1 \in M_1 \wedge \cdots \wedge x_n \in M_n \big\}.$$

Sometimes, n-tuples are called lists. In this case they are written with the square brackets "[" and "]" instead of the angle brackets "⟨" and "⟩" that we are using.

Exercise 3: Assume that M and N are finite sets. How can the expression $\mathit{card}(M \times N)$ be reduced to an expression containing the expressions $\mathit{card}(M)$ and $\mathit{card}(N)$? $\diamond$

2.10 Equality of Sets

We have now presented all the methods that we will use in this lecture in order to construct sets. Next, we discuss the notion of equality of two sets. As a set is solely defined by its members, the question of the equality of two sets is governed by the axiom of extensionality:

Two sets are equal if and only if they have the same elements.

Mathematically, we can capture the axiom of extensionality through the formula

$$M = N \;\leftrightarrow\; \forall x : (x \in M \leftrightarrow x \in N)$$

An important consequence of this axiom is the fact that the order in which the elements are listed in a set does not matter. For example, we have

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

because both sets contain the same elements. Similarly, we have

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

because both these sets contain the elements 1, 2, and 3. It does not matter how often we list these elements when defining a set: An object x either is or is not an element of a given set M. It does not make sense to say something like "M contains the object x n times".[2]

If two sets are defined by explicitly enumerating their elements, the question whether these sets are equal is trivial to decide. However, if a set is defined using the axiom of specification, then it can be very difficult to decide whether this set is equal to another set. For example, it has been shown that

$$\{n \in \mathbb{N}^* \mid \exists x, y, z \in \mathbb{N}^* : x^n + y^n = z^n\} = \{1,2\}.$$

However, the proof of this equation is very difficult because this equation is equivalent to Fermat's conjecture. This conjecture was formulated in 1637 by Pierre de Fermat. It took mathematicians more than three centuries to come up with a rigorous proof that validates this conjecture: In 1994 Andrew Wiles and Richard Taylor were able to do this. There are some similar conjectures concerning the equality of sets that are still open mathematical problems.

2.11 Chapter Review

1. What is a set?
2. How is the axiom of comprehension defined? Why can't we use this axiom to define sets?
3. What is the axiom of restricted comprehension?
4. Lists all the methods that have been introduced to define sets.
5. What is the axiom of extensionality?

If you want to develop a deeper understand of set theory, I can highly recommend the book *Set Theory and Related Topics* by Seymour Lipschutz [Lip98].

[2] In the literature, you will find the concept of a multiset. A multiset does not abstract from the number of occurrences of its elements. In this lecture, we will not use multisets.

Kapitel 3

The Programming Language *Python*

We have started our lecture with an introduction to set theory. In my experience, the notions of set theory are difficult to master for many students because the concepts introduced in set theory are quite abstract. Fortunately, there is a programming language that supports sets as a basic data type and thus enables us to experiment to experiment with set theory. This is the programming language Python, which has its own website at `python.org`. By programming in *Python*, students can get acquainted with set theory in a playful manner. Furthermore, as many interesting problems have a straightforward solution as *Python* programs, students can appreciate the usefulness of abstract notions from set theory by programming in *Python*. Furthermore, according to Philip Guo, 8 of the top 10 US universities teach *Python* in their introductory computer science courses.

The easiest way to install python and its libraries is via Anaconda. On many computers, *Python* is already preinstalled. Nevertheless, even on those systems it is easiest to use the Anaconda distribution. The reason is that Anaconda make it very easy to use different versions of python with different libraries. In this lecture, we will be using the version 3.6 of *Python*.

3.1 Starting the Interpreter

My goal is to introduce *Python* via a number of rather simple examples. I will present more advanced features of *Python* in later sections, but this section is intended to provide a first impression of the language.

```
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 12:04:33)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Abbildung 3.1: The *Python* welcome message.

The language *Python* is an interpreted language. Hence, there is no need to compile a program. Instead, *Python* programs can be executed via the interpreter. The interpreter is started by the command:[1]

```
python
```

After the interpreter is started, the user sees the output that is shown in Figure 3.1 on page 12. The string "»>" is the prompt. It signals that the interpreter is waiting for input. If we input the string

[1] While I am usually in the habit of terminating every sentence with either a full stop, a question mark or an exclamation mark, I refrain from doing so when the sentence ends in a *Python* command that is shown on a separate line. The reason is that I want to avoid confusion as it can otherwise be hard to understand which part of the line is the command that has to be typed verbatim.

```
1 + 2
```

and press enter, we get the following output:

```
3
>>>
```

The interpreter has computed the sum $1 + 2$, returned the result, and prints another prompt waiting for more input. Formally, the command "`1 + 2`" is a script. Of course, this is a very small script as it consists only of a single expression. The command

```
exit()
```

terminates the interpreter. The nice thing about *Python* is the we can run *Python* even in a browser in so called Jupyter notebooks. If you have installed *Python* by means of the Anaconda distribution, then you already have installed Jupyter. The following subsection contains the jupyter notebook `Introduction.ipynb`. You should download this notebook from my github page and try the examples on your own computer. Of course, for this to work you first have to install jupyter.

3.2 An Introduction to *Python*

This *Python* notebook gives a short introduction to *Python*. We will start with the basics but as the main goal of this introduction is to show how *Python* supports sets we will quickly move to more advanced topics. In order to show of the features of *Python* we will give some examples that are not fully explained at the point where we introduce them. However, rest assured that they will be explained eventually.

3.2.1 Evaluating expressions

As Python is an interactive language, expressions can be evaluated directly. In a Jupyter notebook we just have to type Ctrl-Enter in the cell containing the expression. Instead of Ctrl-Enter we can also use Shift-Enter.

```
In [1]: 1 + 2
```

```
Out[1]: 3
```

In *Python*, the precision of integers is not bounded. Hence, the following expression does not cause an overflow.

```
In[2]:1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19*20*21*22*23*24*25
```

```
Out[2]: 15511210043330985984000000
```

The next cell in this notebook shows how to compute the factorial of 1000, i.e. it shows how to compute the product

$$1000! = 1 * 2 * 3 * \cdots * 998 * 999 * 1000$$

It uses some advanced features from functional programming that will be discussed at a later stage of this introduction.

```
In [3]: import functools

        functools.reduce(lambda x, y: (x*y), range(1, 1001))
```

```
Out[3]:
402387260077093773543702433923003985719374864210714632543799910429938512
398629020592044208486969404800479988610197196058631666872994808558901323
829669944590997424504087073759918823627727188732519779505950995276120874
975462497043601418278094646496291056393887437886487337119181045825783647
849977012476632889835955735432513185323958463075557409114262417474349347
553428646576611667797396668820291207379143853719588249808126867838374559
731746136085379534524221586593201928090878297308431392844403281231558611
036976801357304216168747609675871348312025478589320767169132448426236131
412508780208000261683151027341827977704784635868170164365024153691398281
264810213092761244896359928705114964975419909342221566832572080821333186
116811553615836546984046708975602900950537616475847728421889679646244945
160765353408198901385442487984959953319101723355556602139450399736280750
137837615307127761926849034352625200015888535147331611702103968175921510
907788019393178114194545257223865541461062892187960223838971476088506276
862967146674697562911234082439208160153780889893964518263243671616762179
168909779911903754031274622289988005195444414282012187361745992642956581
746628302955570299024324153181617210465832036786906117260158783520751516
284225540265170483304226143974286933061690897968482590125458327168226458
066526769958652682272807075781391858178889652208164348344825993266043367
660176999612831860788386150279465955131156552036093988180612138558600301
435694527224206344631797460594682573103790084024432438465657245014402821
885252470935190620929023136493273497565513958720559654228749774011413346
962715422845862377387538230483865688976461927383814900140767310446640259
899490222221765904339901886018566526485061799702356193897017860040811889
729918311021171229845901641921068884387121855646124960798722908519296819
372388642614839657382291123125024186649353143970137428531926649875337218
940694281434118520158014123344828015051399694290153483077644569099073152
433278288269864602789864321139083506217095002597389863554277196742822248
757586765752344220207573630569498825087968928162753848863396909959826280
956121450994871701244516461260379029309120889086942028510640182154399457
156805941872748998094254742173582401063677404595741785160829230135358081
840096996372524230560855903700624271243416909004153690105933983835777939
410970027753472000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
```

The following command will stop the interpreter if executed. It is not useful inside a Jupyter notebook. Hence, the next line should not be evaluated. Therefore, I have put a comment character "#" in the first column of this line.

However, if you do remove the comment character and then evaluate the line, nothing bad will happen as the interpreter is just restarted by Jupyter.

```
In [4]: # exit()
```

In order to write something to the screen, we can use the function print. This function can print objects of any type. In the following example, this function prints a string. In *Python* any character sequence enclosed in single quotes is string.

```
In [5]: print('Hello, World!')

Hello, World!
```

Instead of using single quotes we can also use double quotes as seen in the next example.

```
In [6]: print("Hello, World!")

Hello, World!
```

The function print accepts any number of arguments. For example, to print the string "36 * 37 / 2 = ff-ollowed by the value of the expression $36 \cdot 37/2$ we can use the following print statement:

```
In [7]: print("36 * 37 / 2 =", 36 * 37 // 2)

36 * 37 / 2 = 666
```

In the expression "36 * 37 // 2" we have used the operator "//" in order to enforce integer division. If we had used the operator "/" instead, *Python* would have used floating point division and therefore would have printed the floating point number 666.0 instead of the integer 666.

```
In [8]: print("36 * 37 / 2 =", 36 * 37 / 2)

36 * 37 / 2 = 666.0
```

The following script reads a natural number n and computes the sum $\sum_{i=1}^{n} i$.

1. The function `input` prompts the user to enter a string.
2. This string is then converted into an integer using the function `int`.
3. Next, the set `s` is created such that
$$\texttt{s} = \{1, \cdots, n\}.$$
The set `s` is constructed using the function `range`. A function call of the form `range`$(a, b+1)$ returns a generator that produces the natural numbers from a to b. By using this generator as an argument to the function `set`, a set is created that contains all the natural number starting from a upto and including b. The precise mechanics of generators will be explained later.
4. The `print` statement uses the function `sum` to add up all the elements of the set `s` and print the resulting sum.

```
In [9]: n = input('Type a natural number and press return: ')
        n = int(n)
        s = set(range(1, n+1))
        print('The sum 1 + 2 + ... + ', n, ' is equal to ', sum(s), '.', sep= '')

Type a natural number and press return: 36
The sum 1 + 2 + ... + 36 is equal to 666.
```

The following example shows how functions can be defined in *Python*. The function `sum`(n) is supposed to compute the sum of all the numbers in the set $\{1, \cdots, n\}$. Therefore, we have

$$\texttt{sum}(n) = \sum_{i=1}^{n} i.$$

The function sum is defined recursively. The recursive implementation of the function sum can best by understood if we observe that it satisfies the following two equations:

1. $\texttt{sum}(0) = 0$,
2. $\texttt{sum}(n) = \texttt{sum}(n-1) + n$ provided that $n > 0$.

```
In [10]: def sum(n):
             if n == 0:
                 return 0
             return sum(n-1) + n
```

Let us discuss the implementation of the function sum line by line:

1. The keyword `def` starts the definition of the function. It is followed by the name of the function that is defined. The name is followed by the list of the parameters of the function. This list is enclosed in parentheses. If there is more than one parameter, the parameters have to be separated by commas. Finally, there needs to be a colon at the end of the first line.
2. The body of the function is indented. **Contrary** to most other programming languages, *Python* is space sensitive.

   The first statement of the body is a conditional statement, which starts with the keyword `if`. The keyword is followed by a test. In this case we test whether the variable n is equal to the number 0. Note that this test is followed by a colon.
3. The next line contains a `return` statement. Note that this statement is again indented. All statements indented by the same amount that follow an `if`-statement are considered to be the body of this `if`-statement, i.e. they get executed if the test of the `if`-statement is true. In this case the body contains only a single statement.
4. The last line of the function definition contains the recursive invocation of the function `sum`.

Using the function sum, we can compute the sum $\sum_{i=1}^{n} i$ as follows:

```
In [11]: n     = int(input("Enter a natural number: "))
         total = sum(n)
         if n > 2:
             print("0 + 1 + 2 + ... + ", n, " = ", total, sep='')
         else:
             print(total)
```

```
Enter a natural number: 100
0 + 1 + 2 + ... + 100 = 5050
```

3.2.2 Sets in *Python*

Python supports sets as a **native** datatype. This is one of the reasons that have lead me to choose *Python* as the programming language for this course. To get a first impression how sets are handled in *Python*, let us define two simple sets A and B and print them:

```
In [12]: A = {1, 2, 3}
         B = {2, 3, 4}
         print('A = ', A, ', B = ', B, sep='')
```

```
A = {1, 2, 3}, B = {2, 3, 4}
```

The last argument `sep=''` prevents the print statement from separating its arguments with space characters. When defining the empty set, there is a caveat, as we cannot define the empty set using the expression `{}`. The reason is that this expression creates the empty dictionary instead. (We will discuss the data type of dictionaries later.) To define the empty set, we therefore have to use the following expression:

```
In [13]: set()
```

```
Out[13]: set()
```

Note that the empty set is also printed as set() in *Python* and not as {}. Next, let us compute the union $A \cup B$. This is done using the function `union`.

```
In [14]: A.union(B)

Out[14]: {1, 2, 3, 4}
```

As the function union really acts like a method, you might suspect that it does change its first argument. Fortunately, this is not the case, A is unchanged as you can see in the next line:

```
In [15]: A

Out[15]: {1, 2, 3}
```

To compute the intersection $A \cap B$, we use the function `intersection`:

```
In [16]: A.intersection(B)

Out[16]: {2, 3}
```

Again A is not changed.

```
In [17]: A

Out[17]: {1, 2, 3}
```

The difference $A \backslash B$ is computed using the operator "-":

```
In [18]: A - B

Out[18]: {1}
```

It is easy to test whether $A \subseteq B$ holds:

```
In [19]: A <= B

Out[19]: False
```

Testing whether an object x is an element of a set M, i.e. to test, whether $x \in M$ holds is straightforward:

```
In [20]: 1 in A

Out[20]: True
```

On the other hand, the number 1 is not an element of the set B, i.e. we have $1 \notin B$:

```
In [21]: 1 not in B

Out[21]: True
```

3.2.3 Defining Sets via Selection and Images

Remember that we can define subsets of a given set M via the axiom of selection. If p is a property such that for any object x from the set M the expression $p(x)$ is either True or False, the subset of all those elements of M such that $p(x)$ is True can be defined as

$$\{x \in M \mid p(x)\}.$$

For example, if M is the set $\{1, \cdots, 100\}$ and we want to compute the subset of this set that contains all numbers from M that are divisible by 7, then this set can be defined as

$$\{x \in M \mid x \,\%\, 7 == 0\}.$$

In *Python*, the definition of this set can be given as follows:

```
In [22]: M = set(range(1, 101))
         { x for x in M if x % 7 == 0 }

Out[22]: {7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98}
```

In general, in *Python* the set

$$\{x \in M \mid p(x)\}$$

is computed by the expression

$$\{\ x \texttt{ for } x \texttt{ in } M \texttt{ if } p(x)\ \}.$$

Image sets can be computed in a similar way. If f is a function defined for all elements of a set M, the image set

$$\{f(x) \mid x \in M\}$$

can be computed in *Python* as follows:

$$\{\ f(x) \texttt{ for } x \texttt{ in } M\ \}.$$

For example, the following expression computes the set of all squares of numbers from the set $\{1, \cdots, 10\}$:

```
In [23]: M = set(range(1,11))
         { x*x for x in M }

Out[23]: {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

The computation of image sets and selections can be combined. If M is a set, p is a property such that $p(x)$ is either True or False for elements of M, and f is a function such that $f(x)$ is defined for all $x \in M$ then we can compute set

$$\{f(x) \mid x \in M \wedge p(x)\}$$

of all images $f(x)$ from those $x \in M$ that satisfy the property $p(x)$ via the expression

$$\{\ f(x) \texttt{ for } x \texttt{ in } M \texttt{ if } p(x)\ \}.$$

For example, to compute the set of those squares of numbers from the set $\{1, \cdots, 10\}$ that are even we can write

```
In [24]: M = set(range(1,11))
         { x*x for x in M if x % 2 == 0 }

Out[24]: {4, 16, 36, 64, 100}
```

We can iterate over more than one set. For example, let us define the set of all products $p \cdot q$ of numbers p and q from the set $\{2, \cdots, 10\}$, i.e. we intend to define the set

$$\{p \cdot q \mid p \in \{2, \cdots, 10\} \wedge q \in \{2, \cdots, 10\}\}.$$

In *Python*, this set is defined as follows:

```
In [25]: print({ p * q for p in range(2,11) for q in range(2,11) })
```

```
{4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 24, 25, 27, 28, 30, 32, 35,
 36, 40, 42, 45, 48, 49, 50, 54, 56, 60, 63, 64, 70, 72, 80, 81, 90, 100}
```

We can use this set to compute the set of prime numbers. After all, the set of prime numbers is the set of all those natural numbers bigger than 1 that can not be written as a proper product, that is a number x is prime if

1. x is bigger than 1 and
2. there are no natural numbers x and y both bigger than 1 such that $x = p * q$ holds.

More formally, the set $\mathbb{P}$ of prime numbers is defined as follows:

$$\mathbb{P} = \{x \in \mathbb{N} \mid x > 1 \wedge \neg\exists p, q \in \mathbb{N} : (x = p \cdot q \wedge p > 1 \wedge q > 1)\}.$$

Hence the following code computes the set of all primes less than 100:

```
In [26]: s = set(range(2,100))
         print(s - { p * q for p in s for q in s })
```

```
{ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
  67, 71, 73, 79, 83, 89, 97
}
```

An alternative way to compute primes works by noting that a number p is prime iff there is no number t other than 1 and p that divides the number p. The function dividers given below computes the set of all numbers dividing a given number p evenly:

```
In [27]: def dividers(p):
             "Compute the set of numbers that divide the number p."
             return { t for t in range(1, p+1) if p % t == 0 }

         n      = 100;
         primes = { p for p in range(2, n) if dividers(p) == {1, p} }
         print(primes)
```

```
{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97}
```

3.2.4 Computing the Power Set

Unfortunately, there is no operator to compute the power set 2^M of a given set M. Since the power set is needed frequently, we have to implement a function `power` to compute this set ourselves. The easiest way to compute the power set 2^M of a set M is to implement the following recursive equations:

1. The power set of the empty set contains only the empty set:

$$2^{\{\}} = \{\{\}\}$$

2. If a set M can be written as $M = C \cup \{x\}$, where the element x does not occur in the set C, then the power set 2^M consists of two sets:

- Firstly, all subsets of C are also subsets of M.
- Secondly, if A is a subset of C, then the set $A \cup \{x\}$ is also a subset of M.

If we combine these parts we get the following equation:

$$2^{C \cup \{x\}} = 2^C \cup \{A \cup \{x\} \mid A \in 2^C\}$$

But there is another problem: In *Python* we can't create a set that has elements that are sets themselves! The reason is that in *Python* sets are implemented via hash tables and therefore the elements of a set need to be hashable. (The notion of an element being hashable will be discussed in more detail in the lecture on Algorithms.) However, sets are mutable and mutable objects are not hashable. Fortunately, there is a workaround: *Python* provides the data type of frozen sets. These sets behave like sets but are are lacking certain functions that modify sets and hence are unmutable. So if we use frozen sets as elements of the power set, we can compute the power set of a given set. The function `power` given below shows how this works.

```
In [28]: def power(M):
             "This function computes the power set of the set M."
             if M == set():
                 return { frozenset() }
             else:
                 C  = set(M)  # C is a copy of M as we don't want to change the set M
                 x  = C.pop() # pop removes the element x from the set C
                 P1 = power(C)
                 P2 = { A.union({x}) for A in P1 }
                 return P1.union(P2)
```

```
In [29]: power(A)
```

```
Out[29]: {frozenset(),
          frozenset({3}),
          frozenset({1}),
          frozenset({2}),
          frozenset({1, 2}),
          frozenset({2, 3}),
          frozenset({1, 3}),
          frozenset({1, 2, 3})}
```

Let us print this in a more readable way. To this end we implement a function prettify that turns a set of frozensets into a string that looks like a set of sets.

```
In [30]: def prettify(M):
             """Turn the set of frozen sets M into a string that looks like a set of sets.
                M is assumed to be the power set of some set.
             """
             result = "{{}, "   # The emepty set is always an element of a power set.
             for A in M:
                 if A == set(): # The empty set has already been taken care of.
                     continue
                 result += str(set(A)) + ", " # A is converted from a frozen set to a set
             result = result[:-2] # remove the trailing substring ", "
             result += "}"
             return result
```

```
In [31]: prettify(power(A))
```

```
Out[31]: '{{}, {3}, {1, 2}, {2, 3}, {1}, {1, 3}, {1, 2, 3}, {2}}'
```

3.2.5 Pairs and Cartesian Products

In *Python*, pairs can be created by enclosing the components of the pair in parentheses. For example, to compute the pair $\langle 1, 2 \rangle$ we can write:

```
In [32]: (1,2)

Out[32]: (1, 2)
```

It is not even necessary to enclose the components of a pair in parentheses. For example, to compute the pair $\langle 1, 2 \rangle$ we can use the following expression:

```
In [33]: 1, 2

Out[33]: (1, 2)
```

The Cartesian product $A \times B$ of two sets A and B can now be computed via the following expression:

$$\{\ (x, y)\ \texttt{for}\ x\ \texttt{in}\ A\ \texttt{for}\ y\ \texttt{in}\ B\ \}$$

For example, as we have defined A as $\{1, 2, 3\}$ and B as $\{2, 3, 4\}$, the Cartesian product of A and B is computed as follows:

```
In [34]: { (x,y) for x in A for y in B }

Out[34]: {(1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 2), (3, 3), (3, 4)}
```

3.2.6 Tuples

The notion of a tuple is a generalization of the notion of a pair. For example, to compute the tuple $\langle 1, 2, 3 \rangle$ we can use the following expression:

```
In [35]: (1, 2, 3)

Out[35]: (1, 2, 3)
```

Longer tuples can be build using the function range in combination with the function tuple:

```
In [36]: tuple(range(1, 11))

Out[36]: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Tuples can be concatenated using the operator "+":

```
In [37]: T1 = (1, 2, 3)
         T2 = (4, 5, 6)
         T3 = T1 + T2
         T3

Out[37]: (1, 2, 3, 4, 5, 6)
```

The length of a tuple is computed using the function `len`:

```
In [38]: len(T3)

Out[38]: 6
```

The components of a tuple can be extracted using square brackets. Not that the first component actually has the index 0! This is similar to the behaviour of arrays in the programming language C.

```
In [39]: print("T3[0] =", T3[0])
         print("T3[1] =", T3[1])
         print("T3[2] =", T3[2])

T3[0] = 1
T3[1] = 2
T3[2] = 3
```

If we use negative indices, then we index from the back of the tuple, as shown in the following example:

```
In [40]: print("T3[-1] =", T3[-1]) # last element
         print("T3[-2] =", T3[-2]) # penultimate element
         print("T3[-3] =", T3[-3]) # third last element

T3[-1] = 6
T3[-2] = 5
T3[-3] = 4

In [41]: T3

Out[41]: (1, 2, 3, 4, 5, 6)
```

The slicing operator extracts a subtuple form a given tuple. If L is a tuple and a and b are natural numbers such that $a \leq b$ and $a, b \in \{0, \texttt{len}(L)\}$, then the syntax of the slicing operator is as follows:

$$L[a : b]$$

The expression $L[a : b]$ extracts the subtuple that starts with the element $L[a]$ up to and excluding the element $L[b]$. The following shows an example:

```
In [42]: L = tuple(range(1,11))
         L[2:6]

Out[42]: (3, 4, 5, 6)
```

Slicing works with negative indices, too:

```
In [43]: L[2:-2]

Out[43]: (3, 4, 5, 6, 7, 8)
```

3.2.7 Lists

Next, we discuss the data type of lists. Lists are a lot like tuples, but in contrast to tuples, lists are mutatable, i.e. we can change lists. To construct a list, we use square backets:

```
In [44]: L = [1,2,3]
         L

Out[44]: [1, 2, 3]
```

To change the first element of a list, we can use the index operator:

```
In [45]: L[0] = 7
         L

Out[45]: [7, 2, 3]
```

This last operation would not be possible if L had been a tuple instead of a list. Lists support concatenation in the same way as tuples:

```
In [46]: [1,2,3] + [4,5,6]

Out[46]: [1, 2, 3, 4, 5, 6]
```

The function `len` computes the length of a list:

```
In [47]: len([4,5,6])

Out[47]: 3
```

Lists and tuples both support the functions `max` and `min`. The expression `max`(L) computes the maximum of all the elements of the list (or tuple) L, while `min`(L) computes the smallest element of L.

```
In [48]: max([1,2,3])

Out[48]: 3

In [49]: min([1,2,3])

Out[49]: 1
```

3.2.8 Boolean Operators

In *Python*, the Boolean values are written as `True` and `False`.

```
In [50]: True

Out[50]: True

In [51]: False

Out[51]: False
```

These values can be combined using the Boolean operator $\wedge$, $\vee$, and $\neg$. In *Python*, these operators are denoted as `and`, `or`, and not. The following table shows how the operator `and` is defined:

```
In [52]: B = (True, False)
         for x in B:
             for y in B:
                 print(x, 'and', y, '=', x and y)

True and True = True
True and False = False
False and True = False
False and False = False
```

The disjunction of two Boolean values is only `False` if both values are `False`:

```
In [53]: for x in B:
             for y in B:
                 print(x, 'or', y, '=', x or y)
```

```
True or True = True
True or False = True
False or True = True
False or False = False
```

Finally, the negation operator `not` works as expected:

```
In [54]: for x in B:
             print('not', x, '=', not x)
```

```
not True = False
not False = True
```

Boolean values are created by comparing numbers using the following comparison operators:

1. a == b is true iff a is equal to b.
2. a != b is true iff a is different from b.
3. a < b is true iff a is less than b.
4. a <= b is true iff a is less than or equal to b.
5. a >= b is true iff a is bigger than or equal to b.
6. a > b is true iff a is bigger than b.

```
In [55]: 1 == 2
```

```
Out[55]: False
```

```
In [56]: 1 < 2
```

```
Out[56]: True
```

```
In [57]: 1 <= 2
```

```
Out[57]: True
```

```
In [58]: 1 > 2
```

```
Out[58]: False
```

```
In [59]: 1 >= 2
```

```
Out[59]: False
```

Comparison operators can be chained as shown in the following example:

```
In [60]: 1 < 2 < 3
```

```
Out[60]: True
```

Python supports the universal quantifier $\forall$ (read: *for all*). If L is a list of Boolean values, then we can check whether all elements of L are true by writing

`all`(L)

For example, to check whether all elements of a list L are even we can write the following:

```
In [61]: L = [2, 4, 6]
         all([x % 2 == 0 for x in L])

Out[61]: True
```

3.2.9 Control Structures

First of all, *Python* supports branching statements. The following example is taken from the *Python* tutorial at https://python.org:

```
In [62]: x = int(input("Please enter an integer: "))
         if x < 0:
            print('The number is negative!')
         elif x == 0:
            print('The number is zero.')
         elif x == 1:
            print("It's a one.")
         else:
            print("It's more than one.")

Please enter an integer: 42
It's more than one.
```

Loops can be used to iterate over sets, lists, tuples, or generators. The following example prints the numbers from 1 to 10.

```
In [63]: for x in range(1, 11):
             print(x)

1
2
3
4
5
6
7
8
9
10
```

The same can be achieved with a `while` loop:

```
In [64]: x = 1
         while x <= 10:
             print(x)
             x += 1
```

The following program computes the prime numbers according to an algorithm given by Eratosthenes.

1. We set n equal to 100 as we want to compute the set all prime numbers less or equal that 100.
2. `primes` is the list of numbers from 0 upto n, i.e. we have initially

   $$\texttt{primes} = [0, 1, 2, \cdots, n]$$

   Therefore, we have

   $$\texttt{primes}[i] = i \quad \text{for all } i \in \{0, 1, \cdots, n\}.$$

   The idea is to set `primes[`i`]` to zero iff i is a proper product of two numbers.
3. To this end we iterate over all i and j from the set $\{2, \cdots, n\}$ and set the product $\texttt{primes}[i * j]$ to zero. This is achieved by the two `for` loops below.
4. Note that we have to check that the product $i * j$ is not bigger than n for otherwise we would get an out of range error when trying to assign `primes[i*j]`.
5. After the iteration, all non-prime elements greater than one of the list primes have been set to zero.
6. Finally, we compute the set of primes by collecting those elements that have not been set to 0.

```
In [65]: n      = 100
         primes = list(range(0, n+1))
         for i in range(2, n+1):
             for j in range(2, n+1):
                 if i * j <= n:
                     primes[i * j] = 0
         print(primes)
         print({ i for i in range(2, n+1) if primes[i] != 0 })
```

```
[0, 1, 2, 3, 0, 5, 0, 7, 0, 0, 0, 11, 0, 13, 0, 0, 0, 17, 0, 19, 0, 0, 0, 23,
 0, 0, 0, 0, 0, 29, 0, 31, 0, 0, 0, 0, 0, 37, 0, 0, 0, 41, 0, 43, 0, 0, 0, 47,
 0, 0, 0, 0, 0, 53, 0, 0, 0, 0, 0, 59, 0, 61, 0, 0, 0, 0, 0, 67, 0, 0, 0, 71,
 0, 73, 0, 0, 0, 0, 0, 79, 0, 0, 0, 83, 0, 0, 0, 0, 0, 89, 0, 0, 0, 0, 0, 0,
 0, 97, 0, 0, 0]
{ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
  73, 79, 83, 89, 97
}
```

The algorithm given above can be improved by using the following observations:

1. If a number x can be written as a product $a * b$, then at least one of the numbers a or b has to be less than $\sqrt{x}$. Therefore, the `for` loop below iterates as long as $i \leq \sqrt{x}$. The function `ceil` is needed to cast the square root of x to a natural number. In order to use the functions `sqrt` and `ceil` we have to import them from the module `math`. This is done in line 1 of the program shown below.

2. When we iterate over j in the inner loop, it is sufficient if we start with $j = i$ since all products of the form $i * j$ where $j < i$ have already been eliminated at the time, when the multiples of i had been eliminated.
3. If `primes[i] = 0`, then i is not a prime and hence it has to be a product of two numbers a and b both of which are smaller than i. However, since all the multiples of a and b have already been eliminated, there is no point in eliminating the multiples of i since these are also mulples of both a and b and hence have already been eliminated. Therefore, if `primes[i] = 0` we can immediately jump to the next value of i. This is achieved by the `continue` statement in line 7 below.

The program shown below is easily capable of computing all prime numbers less than a million.

```
In [66]: from math import sqrt, ceil

         n = 1000
         primes = list(range(n+1))
         for i in range(2, ceil(sqrt(n))):
             if primes[i] == 0:
                 continue
             j = i
             while i * j <= n:
                 primes[i * j] = 0
                 j += 1;
         print({ i for i in range(2, n+1) if primes[i] != 0 })
```

```
{ 2, 3, 5, 7, 521, 11, 523, 13, 17, 19, 23, 29, 541, 31, 547, 37, 41, 43, 557,
  47, 563, 53, 569, 59, 571, 61, 577, 67, 71, 73, 587, 79, 593, 83, 599, 89,
  601, 607, 97, 101, 613, 103, 617, 107, 619, 109, 113, 631, 127, 641, 131,
  643, 647, 137, 139, 653, 659, 149, 661, 151, 157, 673, 163, 677, 167, 683,
  173, 179, 691, 181, 701, 191, 193, 197, 709, 199, 719, 211, 727, 733, 223,
  227, 739, 229, 743, 233, 239, 751, 241, 757, 761, 251, 257, 769, 773, 263,
  269, 271, 787, 277, 281, 283, 797, 293, 809, 811, 307, 821, 311, 823, 313,
  827, 317, 829, 839, 331, 337, 853, 857, 347, 859, 349, 863, 353, 359, 877,
  367, 881, 883, 373, 887, 379, 383, 389, 907, 397, 911, 401, 919, 409, 929,
  419, 421, 937, 941, 431, 433, 947, 439, 953, 443, 449, 967, 457, 971, 461,
  463, 977, 467, 983, 479, 991, 997, 487, 491, 499, 503, 509
}
```

3.2.10 Numerical Functions

Python provides all of the mathematical functions that you have come to learn at school. A detailed listing of these functions can be found at https://docs.python.org/3.6/library/math.html. We just show the most important functions and constants. In order to make the module `math` available, we use the following `import` statement:

```
In [67]: import math
```

The mathematical constant Pi, which is most often written as π, is available as `math.pi`.

```
In [68]: math.pi
```

```
Out[68]: 3.141592653589793
```

The sine function is called as follows:

```
In [69]: math.sin(math.pi/6)
```

```
Out[69]: 0.49999999999999994
```

The cosine function is called as follows:

```
In [70]: math.cos(0.0)

Out[70]: 1.0
```

The tangent function is called as follows:

```
In [71]: math.tan(math.pi/4)

Out[71]: 0.9999999999999999
```

The arc sine, arc cosine, and arc tangent are called by prefixing the character 'a' to the name of the function as seen below:

```
In [72]: math.asin(1.0)

Out[72]: 1.5707963267948966

In [73]: math.acos(1.0)

Out[73]: 0.0

In [74]: math.atan(1.0)

Out[74]: 0.7853981633974483
```

Euler's number e can be computed as follows:

```
In [75]: math.e

Out[75]: 2.718281828459045
```

The exponential function $\exp(x) := e^x$ is computed as follows:

```
In [76]: math.exp(1)

Out[76]: 2.718281828459045
```

The natural logarithm $\ln(x)$, which is defined as the inverse function of the function $\exp(x)$, is called `log` (instead of `ln`):

```
In [77]: math.log(math.e * math.e)

Out[77]: 2.0
```

The square root $\sqrt{x}$ of a number x is computed using the function `sqrt`:

```
In [78]: math.sqrt(2)

Out[78]: 1.4142135623730951
```

3.2.11 Selection Sort

In order to see a practical application of the concepts discussed so far, we present a sorting algorithm that is known as selection sort. This algorithm sorts a given list `L` and works as follows:

1. If `L` is empty, `sort(L)` is also empty:

   `sort([])` = `[]`.

2. Otherwise, we first compute the minimum of `L`. Clearly, the minimum needs to be the first element of the sorted list. We remove this minimum from `L`, sort the remaining elements recursively, and finally attach the minimum at the front of this list:

   `sort(L)` = `[min(L)] + sort([`$x \in$ `L|`$x \neq$ `min(`L`)])`.

Figure 3.2 on page 29 shows the program `min-sort.py` that implements selection sort in *Python*.

```
1  def minSort(L):
2      if L == []:
3          return []
4      m = min(L)
5      return [m] + minSort([x for x in L if x != m])
6
7  L = [ 2, 13, 5, 13, 7, 2, 4 ]
8  print('minSort(', L, ') = ', minSort(L), sep='')
```

Abbildung 3.2: Implementing selection sort in *Python*.

3.3 Loading a Program

The SETLX interpreter can load programs interactively into a running session. If *file* is the base name of a file, then the command

`import` *file*

loads the program from *file*`.py` and executes the statements given in this program. For example, the command

```
import min_sort
```

executes the program shown in Figure 3.2 on page 29. If we want to call a function defined in the file `min_sort.py`, then we have to prefix this function as shown below:

```
min_sort.minSort([2, 13, 5, 13, 7, 2, 4]),
```

i.e. we have to prefix the name of the function that we want to call with the base name of the file defining this function followed by a dot character.

3.4 Strings

Python support strings. Strings are nothing more but sequences of characters. In *Python*, these have to be enclosed either in double quotes or in single quotes. The operator "+" can be used to concatenate strings. For example, the expression

```
"abc" + 'uvw';
```

returns the result

`"abcuvw"`.

Furthermore, a natural number `n` can be multiplied with a string `s`. The expression

```
n * s;
```

returns a string consisting of `n` concatenations of `s`. For example, the result of

```
3 * "abc";
```

is the string `"abcabcabc"`. When multiplying a string with a number, the order of the arguments does not matter. Hence, the expression

```
"abc" * 3
```

also yields the result `"abcabcabc"`. In order to extract substrings from a given string, we can use the same slicing operator that also works for lists and tuples. Therefore, if s is a string and k and l are numbers, then the expression

$s[k..l]$

extracts the substring form s that starts with the $k+1$th character of s and that ends with the lth character. For example, if `s` is defined by the assignment

```
s = "abcdefgh"
```

then the expression `s[2:5]` returns the substring

`"cde"`.

3.5 Computing with Unlimited Precision

Python provides the module fractions that implements rational numbers through the function Fraction that is implemented in this module. We can load this function as follows:

```
In [1]: from fractions import Fraction
```

The function Fraction expects two arguments, the nominator and the denominator. Mathematically, we have

$$\texttt{Fraction}(p,q) = \frac{p}{q}.$$

For example, we can compute the sum $\frac{1}{2} + \frac{1}{3}$ as follows:

```
In [2]: sum = Fraction(1, 2) + Fraction(1, 3)
        print(sum)
```

```
5/6
```

Let us compute Euler's number e. The easiest way to compute e is as inifinite series. We have that

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

Here $n!$ denotes the factorial of n, which is defined as follows:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n.$$

```
In [3]: def factorial(n):
            "compute the factorial of n"
            result = 1
            for i in range(1, n+1):
                result *= i
            return result
```

Let's check that our definition of the factorial works as expected.

```
In [4]: for i in range(10):
            print(i, '! = ', factorial(i), sep='')
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
```

Lets approximate e by the following sum:

$$e = \sum_{i=0}^{n} \frac{1}{i!}$$

Setting $n = 100$ should be sufficient to compute e to a hundred decimal places.

```
In [5]: n = 100
```

```
In [6]: e = 0
        for i in range(n+1):
            e += Fraction(1, factorial(i))
```

Multiply e by 10^{100} and round so that we get the first 100 decimal places of e:

```
In [7]: eTimesBig = e * 10 ** n
        s = str(round(eTimesBig))
```

Insert a "." after the first digit:

```
In [8]: print(s[0], '.', s[1:], sep='')
```

And there we go. Ladies and gentlemen, lo and behold: Here are the first 100 digits of e:

```
2.718281828459045235360287471352662497757247093699959574966967627724076630353547
594571382178525166427
```

3.6 Dictionaries

A *binary relation* R is a subset of the cartesian product of two sets A and B, i.e. if R is a binary relation we have:

$$R \subseteq A \times B$$

A binary relation $R \subseteq A \times B$ is a *functional* relation if and only if we have:

$$\forall x \in A : \forall y_1, y_2 \in B : \big(\langle x, y_1\rangle \in R \wedge \langle x, y_2\rangle \in R \rightarrow y_1 = y_2\big)$$

If R is a fuctional relation, then $R \subseteq A \times B$ can be interpreted as a function

$$f_R : A \rightarrow B$$

that is defined as follows:

$$f_R(x) := y \quad \text{iff} \quad \langle x, y \rangle \in R.$$

In *Python* a functional relation $R \subseteq A \times B$ can be represented as a **dictionary**, provided R is finite. The empty dictionary is defined as follows:

```
In [2]: emptyDict = {}
```

The syntax to define a functional relation R of the form

$$\{\langle x_1, y_1 \rangle, \cdots, \langle x_n, y_n \rangle\}$$

in *Python* is as follows: We have to write

```
{ x1:y1, ... xn:yn }
```

An example will clarify this. The dictionary Number2English maps the first nine numbers to their English names.

```
In [3]: Number2English = { 1:'one', 2:'two', 3:'three', 4:'four', 5:'five',
                           6:'six', 7:'seven', 8:'eight', 9:'nine'
                         }
        Number2English

Out[3]: {1: 'one',
         2: 'two',
         3: 'three',
         4: 'four',
         5: 'five',
         6: 'six',
         7: 'seven',
         8: 'eight',
         9: 'nine'}
```

Here, the numbers $1, \cdots, 9$ are called the *keys* of the dictionary.

We can use the dictionary Number2English as if it were a function: If we write `Number2English[`k`]`, then this expression will return the name of the number k provided $k \in \{1, \cdots, 9\}$.

```
In [4]: Number2English[2]

Out[4]: 'two'
```

The expression Number2English[10] would return an error message, as 10 is not a key of the dictionary Number2English. We can check whether an object is a key of a dictionary by using the operator in as shown below:

```
In [5]: 10 in Number2English

Out[5]: False

In [6]: 7 in Number2English

Out[6]: True
```

We can easily extend our dictionary as shown below:

```
In [7]: Number2English[10] = 'ten'

In [8]: Number2English
```

```
Out[8]: {1: 'one',
         2: 'two',
         3: 'three',
         4: 'four',
         5: 'five',
         6: 'six',
         7: 'seven',
         8: 'eight',
         9: 'nine',
         10: 'ten'}
```

In order to have more fun, let us define a second dictionary.

```
In [9]: Number2Hebrew = { 1:"echad", 2:"shtaim", 3:"shalosh", 4:"arba", 5:"hamesh",
                          6:"shesh", 7:"sheva", 8:"shmone", 9: "tesha", 10: "eser"
                        }
```

Disclaimer: I don't know any Hebrew, I have taken these names from the youtube video at

https://www.youtube.com/watch?v=FBd9QdpqUz0.

Dictionaries can be built via comprehension expressions. Let us demonstrate this be computing the inverse of the dictionary Number2English:

```
In [10]: English2Number = { Number2English[name]:name for name in Number2English }

In [11]: English2Number

Out[11]: {'one': 1,
          'two': 2,
          'three': 3,
          'four': 4,
          'five': 5,
          'six': 6,
          'seven': 7,
          'eight': 8,
          'nine': 9,
          'ten': 10}
```

The example above shows that we can iterate over the keys of a dictionary. Lets use this to build a dictionary that translates the English names of numbers into their Hebrew equivalents:

```
In [12]: English2Hebrew = { name:Number2Hebrew[English2Number[name]] for name in English2Number }

In [13]: English2Hebrew

Out[13]: {'one': 'echad',
          'two': 'shtaim',
          'three': 'shalosh',
          'four': 'arba',
          'five': 'hamesh',
          'six': 'shesh',
          'seven': 'sheva',
          'eight': 'shmone',
          'nine': 'tesha',
          'ten': 'eser'}
```

In order to get the number of entries in a dictionary, we can use the function `len`.

```
In [14]: len(English2Hebrew)

Out[14]: 10
```

If we want to delete an entry from a dictionary, we can use the keyword `del` as follows:

```
In [15]: del Number2English[1]

In [16]: Number2English

Out[16]: {2: 'two',
          3: 'three',
          4: 'four',
          5: 'five',
          6: 'six',
          7: 'seven',
          8: 'eight',
          9: 'nine',
          10: 'ten'}
```

It is important to know that only *immutable* objects can serve as keys in a dictionary. Therefore, number, strings, tuples, or frozensets can be used as keys, but lists or sets can not be used as keys.

Given a dictionary d, the method d.`items`() can be used to iterate over all key-value pairs stored in the dictionary d.

```
In [17]: { pair for pair in English2Hebrew.items() }

Out[17]: {('eight', 'shmone'),
          ('five', 'hamesh'),
          ('four', 'arba'),
          ('nine', 'tesha'),
          ('one', 'echad'),
          ('seven', 'sheva'),
          ('six', 'shesh'),
          ('ten', 'eser'),
          ('three', 'shalosh'),
          ('two', 'shtaim')}
```

This last example shows that the entries in a dictionary are not ordered. In this respect, dictionaries are similar to sets.

3.7 Other References

For reasons of time an space, this lecture has just scratched the surface of what is possible with *Python*. If you want to attain a deeper understanding of *Python*, here are three places that I would recommend:

1. First, there is the official *Python* tutorial, which is available at

   `https://docs.python.org/3.6/tutorial/index.html`.

   Furthermore, there are a number of good books available. I would like to suggest the following two books. Both of these books should be available electronically in our library:

2. *The Quick Python Book* written by Naomi R. Ceder [Ced18] is up to date and gives a concise introduction to *Python*. The book assumes that the reader has some prior programming experience. I would assume that most of our students have the necessary background to feel comfortable with this book.
3. *Learning Python* by Mark Lutz [Lut13] is aimed at the complete novice. It discusses everything in minute detail, albeit at the cost of 1648 pages.

Since *Python* is not the primary objective of these lecture notes, there is no requirement to read either the *Python* tutorial or any of the books mentioned above. The primary objective of these lecture notes is to introduce the main ideas of both propositional logic and predicate logic. *Python* is merely used to illustrate the most important notions from set theory and logic. You should be able to pick up enough knowledge of *Python* by closely inspecting the *Python* programs discussed in these lecture notes.

3.8 Reflection

After having completed this chapter, you should be able to answer the following questions.

1. Which data types are supported in *Python*?
2. What are the different methods to define a set in *Python*?
3. Do you understand how to construct lists via iterator?
4. How can lists be defined in *Python*?
5. How does *Python* support binary relations?
6. How does list slicing and list indexing work?
7. How does a fixed-point algorithm work?
8. What type of control structures are supported in *Python*?

Kapitel 4

Applications and Case Studies

This chapter contains a number of case studies designed to deepen our understanding of *Python*.

4.1 Solving Equations via Fixed-Point Algorithms

Fixed-Point iterations are very important, both in computer science and in mathematics. As a first example, we show how to solve an equation via a fixed point iteration. Suppose we want to solve the equation

$$x = \cos(x).$$

Here, x is a real number that we seek to compute. Figure 4.1 on page 37 shows the graphs of the two functions

$$y = x \quad \text{and} \quad y = \cos(x).$$

Since the graphs of these functions intersect, it is obvious that there exists a value x such that $x = \cos(x)$. Furthermore, from Figure 4.1 it is obvious that this value of x is bigger than 0.6 and less than 0.8.

A simple approach that lets us compute the exact value of x is to use a fixed-point iteration. To this end, we define the sequence $\left(x_n\right)_{n\in\mathbb{N}}$ inductively as follows:

$$x_0 = 0 \quad \text{and} \quad x_{n+1} = \cos(x_n) \quad \text{for all } n \in \mathbb{N}.$$

With the help of the Banach fixed-point theorem[1] it can be shown that this sequence converges to a solution of the equation $x = \cos(x)$, i.e. if we define

$$\bar{x} = \lim_{n\to\infty} x_n,$$

then we have

$$\cos(\bar{x}) = \bar{x}.$$

Figure 4.2 on page 37 shows the program `solve.py` that uses this approach to solve the equation $x = \cos(x)$.

In this program, the iteration stops as soon as the difference between the variables `x` and `old_x` is less that $4 \cdot 10^{-16}$. Here, `x` corresponds to x_{n+1}, while `old_x` corresponds to x_n. Once the values of x_{n+1} and x_n are sufficiently close, the execution of the `while` loop terminates. Fixed-Point-Iteration.ipynb shows a *Jupyter* notebook that implements fixed point iteration.

Figure 4.3 on page 38 shows the program `fixpoint.py`. In this program we have implemented a function `solve` that takes two arguments.

1. `f` is a unary function. The purpose of the `solve` is to compute the solution of the equation

   $$f(x) = x.$$

[1] The Banach fixed-point theorem is discussed in the lecture on differential calculus. This lecture is part of the second semester.

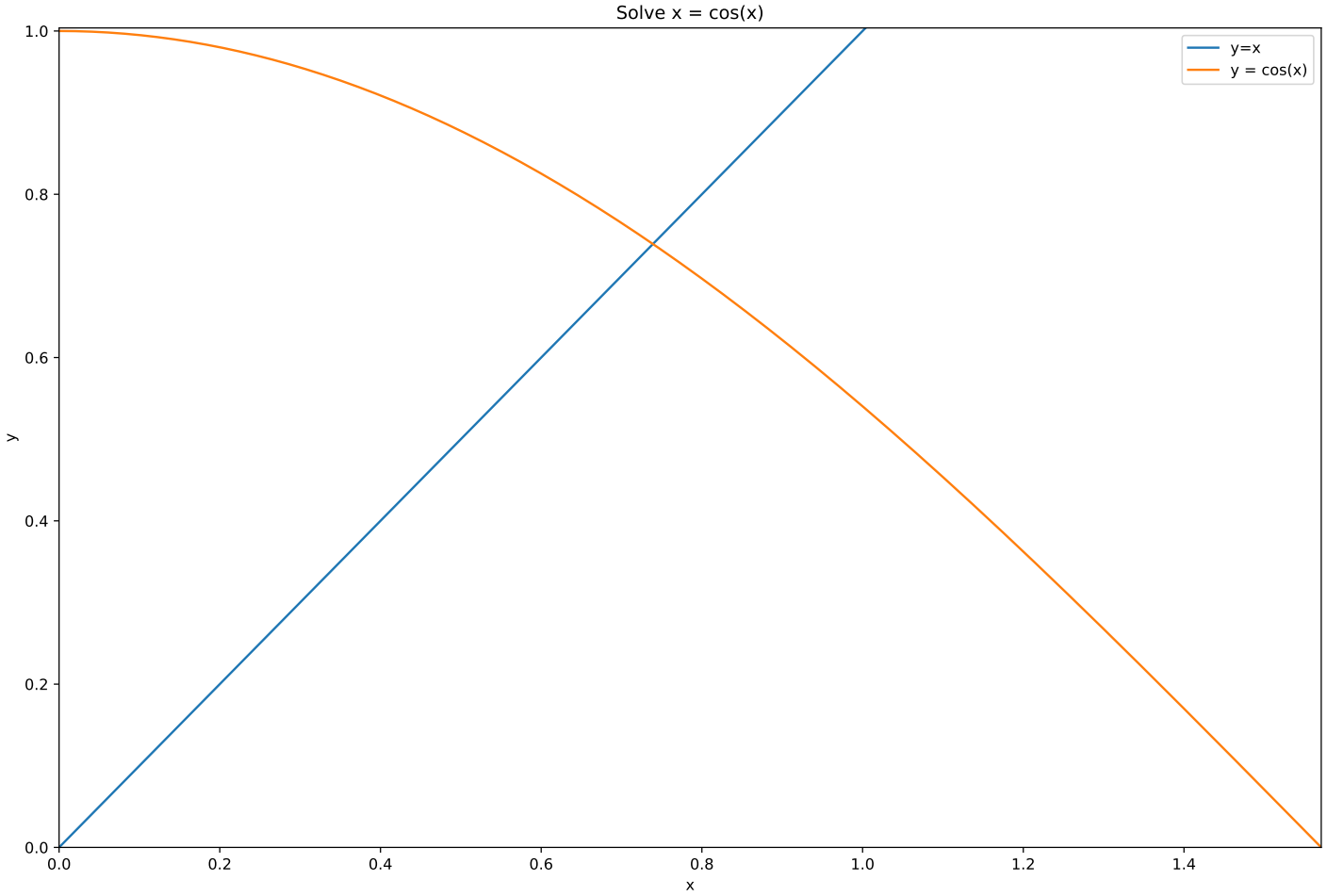

Abbildung 4.1: The functions $y = x$ and $y = cos(x)$.

```
import math

x     = 1.0
old_x = 0.0
i     = 1
while abs(x - old_x) >= 4.0E-16:
    old_x = x
    x = math.cos(x)
    print(f'{i} : {x}')
    i += 1
```

Abbildung 4.2: Solving the equation $x = \cos(x)$ via fixed-point iteration.

```
1  from math import cos
2
3  def solve(f, x0):
4      """
5      Solve the equation f(x) = x using a fixed point iteration.
6      x0 is the start value.
7      """
8      x = x0
9      for n in range(10000):  # at most 10000 iterations
10         oldX = x;
11         x    = f(x);
12         if abs(x - oldX) < 1.0e-15:
13             return x;
14
15 print("solution to x = cos(x): ", solve(cos, 0));
16 print("solution to x = 1/(1+x):", solve(lambda x: 1/(1+x), 0));
```

Abbildung 4.3: A generic implementation of the fixed-point algorithm.

This equation is solved with the help of a fixed-point algorithm.

2. `x0` is used as the initial value for the fixed-point iteration.

Line 11 calls `solve` to compute the solution of the equation $x = \cos(x)$. Line 12 solves the equation

$$x = \frac{1}{1+x}.$$

This equation is equivalent to the quadratic equation $x^2 + x = 1$. Note that we have defined the function $x \mapsto \dfrac{1}{1+x}$ via the expression

`lambda x: 1/(1+x)`.

This expression is called an anonymous function since we haven't given a name to the function.

Remark: The function `solve` is only able to solve the equation $f(x) = x$ if the function f is a contraction mapping. A function $f : \mathbb{R} \to \mathbb{R}$ is called a contraction mapping iff

$$|f(x) - f(y)| < |x - y| \quad \text{for all } x, y \in \mathbb{R}.$$

This notion will be discussed in more detail in the lecture on analysis in the second semester. ◇

4.2 Case Study: Computation of Poker Probabilities

In this short section we are going to show how to compute probabilities for the *Texas Hold'em* variation of poker. Texas Hold'em poker is played with a deck of 52 cards. Every card has a value. This value is an element of the set

$$\mathit{Values} = \{2, 3, 4, 5, 6, 7, 8, 9, 10, \mathit{Jack}, \mathit{Queen}, \mathit{King}, \mathit{Ace}\}.$$

Furthermore, every card has a suit. This suit is an element of the set

$$\mathit{Suits} = \{\clubsuit, \heartsuit, \diamondsuit, \spadesuit\}.$$

These suits are pronounced club, heart, diamond, and spade. As a card is determined by its value and its suit, a card can be represented as a pair $\langle v, s \rangle$, where v denotes the value while s is the suit of the card. Hence, the

set of all cards can be represented as the set

$$\mathit{Deck} = \big\{ \langle v, s \rangle \mid v \in \mathit{Values} \wedge s \in \mathit{Suits} \big\}.$$

At the start of a game of Texas Hold'em, every player receives two cards. These two cards are known as the preflop or the hole. Next, there is a bidding phase where players can bet on their cards. After this bidding phase, the dealer puts three cards open on the table. These three cards are known as flop. Let us assume that a player has been dealt the set of cards

$$\{\langle 3, \clubsuit \rangle, \langle 3, \spadesuit \rangle\}.$$

This set of cards is known as a pocket pair. Then the player would like to know the probability that the flop will contain another card with value 3, as this would greatly increase her chance of winning the game. In order to compute this probability we have to compute the number of possible flops that contain a card with the value 3 and we have to divide this number by the number of all possible flops:

$$\frac{\text{number of flops containing a card with value 3}}{\text{number of all possible flops}}$$

The program poker-triple.py shown in Figure 4.4 performs this computation. We proceed to discuss this program line by line.

```
1  Values = { "2", "3", "4", "5", "6", "7", "8", "9", "T", "J", "Q", "K", "A" }
2  Suits  = { "c", "h", "d", "s" }
3  Deck   = { (v, s) for v in Values for s in Suits }
4  Hole   = { ("3", "c"), ("3", "s") }
5  Rest   = Deck - Hole
6  Flops  = { (k1, k2, k3) for k1 in Rest for k2 in Rest for k3 in Rest
7                          if  len({ k1, k2, k3 }) == 3
8           }
9  Trips  = { f for f in Flops if ("3", "d") in f or ("3", "h") in f }
10 print(len(Trips) / len(Flops))
```

Abbildung 4.4: Computing a probability in poker.

1. In line 1 the set `Values` is defined to be the set of all possible values that a card can take. In defining this set we have made use of the following abbreviations:
   (a) "`T`" is short for "Ten",
   (b) "`J`" is short for "Jack",
   (c) "`Q`" is short for "Queen",
   (d) "`K`" is short for "King", and
   (e) "`A`" is short for "Ace".
2. In line 2 the set `Suits` represents the possible suits of a card. Here, we have used the following abbreviations:
   (a) "`c`" is short for ♣, which is pronounced as club,
   (b) "`h`" is short for ♡, which is pronounced as heart,
   (c) "`d`" is short for ♢, which is pronounced as diamond, and
   (d) "`s`" is short for ♠, which is pronounced as spade.
3. Line 3 defines the set of all cards. This set is stored as the variable `Deck`. Every card is represented as a pair of the form $[v, s]$. Here, v is the value of the card, while s is its suit.

4. Line 4 defines the set `Hole`. This set represents the two cards that have been given to our player.
5. The remaining cards are defined as the variable `Rest` in line 5.
6. Line 6 computes the set of all possible flops. Since the order of the cards in the flop does not matter, we use sets to represent these flops. However, we have to take care that the flop does contain three different cards. Hence, we have to ensure that the three cards `k1`, `k2`, and `k3` that make up the flop satisfy the inequalities

   $$\texttt{k1} \neq \texttt{k2}, \quad \texttt{k1} \neq \texttt{k3}, \quad \text{and} \quad \texttt{k2} \neq \texttt{k3}.$$

   These inequalities are satisfied if and only if the set $\{\texttt{k1}, \texttt{k2}, \texttt{k3}\}$ contains exactly three elements. Hence, when choosing `k1`, `k2`, and `k3` we have to make sure that the condition

   $$\texttt{len}\big(\{\texttt{k1}, \texttt{k2}, \texttt{k3}\} == 3\big)$$

   holds.
7. Line 9 computes the subset `Trips` of those flops that contain at least one card with a value of 3. As the 3 of clubs and the 3 of spades have already been dealt to our player, the only cards with value 3 that are left in the deck are the 3 of diamonds and the 3 of hearts. Therefore, we are looking for those flops that contain one of these two cards.
8. Finally, the probability for obtaining another card with a value of 3 in the flop is computed as the ratio of the number of flops containing a card with a value of 3 to the number of all possible flops.

When we run the program we see that the probability of improving a pocket pair on the flop to trips or better is about 11.8%. A *Jupyter* notebook showcasing this computation outlined above can be fount at Poker.ipynb.

Remark: The method to compute probabilities that has been sketched above only works if the sets that have to be computed are small enough to be retained in memory. If this condition is not satisfied we can use the *Monte Carlo method* to compute the probabilities instead. This method will be discussed in the lecture on algorithms.

4.3 Finding a Path in a Graph

In the following section, I will present an application that is more interesting since it is practically relevant. In order to prepare for this, we will now discuss the problem of finding a path in a directed graph. Abstractly, a graph consists of vertices and edges that connect these vertices. In an application, the vertices could be towns and villages, while the edges would be interpreted as one-way streets connecting these villages. To simplify matters, let us assume for now that the vertices are given as natural numbers. As the edges represent connections between vertices, the edges are represented as pairs of natural numbers. Then, the graph can be represented as the set of its edges, as the set of vertices is implicitly given once the edges are known. To make things concrete, let us consider an example. In this case, the set of edges is called `R` and is defined as follows:

$$\texttt{R} = \big\{\langle 1,2\rangle, \langle 2,3\rangle, \langle 1,3\rangle, \langle 2,4\rangle, \langle 4,5\rangle\big\}.$$

In this graph, the set of vertices is given as

$$\{1,2,3,4,5\}.$$

This graph is shown in Figure 4.5 on page 41. You should note that the connections between vertices that are given in this graph are unidirectional: While there is a connection from vertex 1 to vertex 2, there is no connection from vertex 2 to vertex 1.

The graph given by the relation `R` contains only the direct connections of vertices. For example, in the graph shown in Figure 4.5, there is a direct connection from vertex 1 to vertex 2 and another direct connection from vertex 2 to vertex 4. Intuitively, vertex 4 is reachable from vertex 1, since from vertex 1 we can first reach vertex 2 and from vertex 2 we can then reach vertex 4. However, there is is no direct connection between the vertices 1 and 4. To make this more formal, define a path of a graph R as a list of vertices

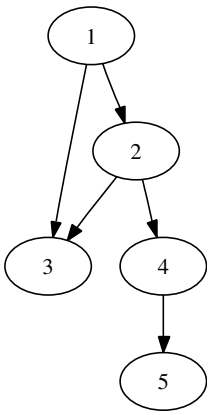

Abbildung 4.5: A simple graph.

$$[x_1, x_2, \cdots, x_n] \quad \text{such that} \quad \langle x_i, x_{i+1} \rangle \in R \quad \text{for all } i = 1, \cdots, n-1.$$

In this case, the path $[x_1, x_2, \cdots, x_n]$ is written as

$$x_1 \mapsto x_2 \mapsto \cdots \mapsto x_n$$

and has the length $n-1$. It is important to note that the length of a path $[x_1, x_2, \cdots, x_n]$ is defined as the number of edges connecting the vertices and not as the number of vertices appearing in the path.

Furthermore, two vertices a and b are said to be connected iff there exists a path

$$[x_1, \cdots, x_n] \quad \text{such that} \quad a = x_1 \quad \text{and} \quad b = x_n.$$

The goal of this section is to develop an algorithm that checks whether two vertices a and b are connected. Furthermore, we want to be able to compute the corresponding path connecting the vertices a and b.

4.3.1 Computing the Transitive Closure of a Relation

We have already noted that a graph can be represented as the set of its edges and hence as a binary relation. A binary relation is defined as a set of pairs. We also need the notion of a relational product: If Q and R are binary relations, then the relational product $Q \circ R$ of Q and R is defined as

$$Q \circ R := \big\{ \langle x, z \rangle \mid \exists y : (\langle x, y \rangle \in Q \wedge \langle y, z \rangle \in R) \big\}.$$

Furthermore, for any $n \in \mathbb{N}^*$ we can define the n-th power R^n of the relation R by induction.

Base Case: $n = 1$.

$$R^1 := R$$

Induction Step: $n \mapsto n + 1$

$$R^{n+1} := R^n \circ R.$$

In order to decide whether there is a path connecting two vertices we have to compute the transitive closure R^+ of a relation R. To understand this notion, we first need to define the concept of transitivity: A relation R is transitive if and only if the following holds:

$$\langle x, y \rangle \in T \wedge \langle y, z \rangle \in T \rightarrow \langle x, z \rangle \in T \quad \text{for all } x, y, z.$$

Now the transitive closure R^+ of a binary relation R is the smallest relation T such that the following conditions hold:

- R is a subset of T, i.e. we have $R \subseteq T$.

- T is transitive.

The lecture on Lineare Algebra gives a prove that the transitive closure R^+ of a binary relation can be computed as follows:

$$R^+ = \bigcup_{n=1}^{\infty} R^n = R^1 \cup R^2 \cup R^3 \cup \cdots$$

Initially, this formula might look intimidating as it suggests an infinite computation. Fortunately, it turns out that we do not have to compute the powers R^n for every $n \in \mathbb{N}$. Let me explain the reason that allows us to cut the computation short.

1. R is the set of direct connections between two vertices.
2. R^2 is the same as $R \circ R$ and this relational product is defined as

   $$R \circ R = \big\{\langle x, z\rangle \mid \exists y\colon (\langle x, y\rangle \in R \wedge \langle y, z\rangle) \in R\big\}.$$

   Hence, $R \circ R$ contains those pairs $\langle x, z\rangle$ that are connected via one intermediate vertex y, i.e. there is a path of the form $x \mapsto y \mapsto z$ that connects x and z. This path has length 2. In general, we can show by induction on n that R^n connect those pairs that are connected by a path of length n. The induction step of this proof runs as follows:

   R^{n+1} is defined as $R^n \circ R$ and therefore we have

   $$R^n \circ R = \{\langle x, z\rangle \mid \exists y\colon \langle x, y\rangle \in R^n \wedge \langle y, z\rangle \in R\}.$$

   As $\langle x, y\rangle \in R^n$, the induction hypothesis guarantees that the vertices x and y are connected by a path of length n. Hence, this path has the form

   $$\underbrace{x \mapsto \cdots \mapsto y}_{\text{path of length } n.}$$

   Adding z at the end of this path will produce the path

   $$x \mapsto \cdots \mapsto y \mapsto z.$$

   This path has a length of $n+1$ and, furthermore, connects x and z. Hence R^{n+1} contains those pairs $\langle x, z\rangle$ that are connected by a path of length $n+1$.

Now the important observation is the following. The set of all vertices is finite. For the arguments sake, let us assume there are k different vertices. But then every path that has a length of k or greater must contain at least one vertex that is visited more than once and hence this path is longer than necessary, i.e. there is a shorter path that connects the same vertices. Therefore, for a finite graph with k vertices, the formula to compute the transitive closure can be simplified as follows:

$$R^+ = \bigcup_{i=1}^{k-1} R^i.$$

While we could use this formula as its stands, it is more efficient to use a fixed-point iteration instead. To this end, we prove that the transitive closure R^+ satisfies the following equation:

$$R^+ = R \cup R^+ \circ R. \tag{4.1}$$

The precedence of the operator $\circ$ is higher than the precedence of the operator $\cup$. Therefore, the expression $R \cup R^+ \circ R$ is equivalent to the expression $R \cup (R^+ \circ R)$. Equation 4.1 can be proven algebraically. We have:

$$
\begin{aligned}
& R \cup R^+ \circ R \\
= \; & R \cup \left(\bigcup_{i=1}^{\infty} R^i \right) \circ R \\
= \; & R \cup \left(R^1 \cup R^2 \cup R^3 \cup \cdots \right) \circ R \\
= \; & R \cup \left(R^1 \circ R \cup R^2 \circ R \cup R^3 \circ R \cup \cdots \right) \\
= \; & R \cup \left(R^2 \cup R^3 \cup R^4 \cup \cdots \right) \\
= \; & R^1 \cup \left(R^2 \cup R^3 \cup R^4 \cup \cdots \right) \\
= \; & \bigcup_{i=1}^{\infty} R^i \\
= \; & R^+.
\end{aligned}
$$

Equation 4.1 can now be used to compute R^+ via a fixed-point iteration. To this end, let us define a sequence of relations $(T_n)_{n \in \mathbb{N}}$ by induction on n:

I.A. $n = 0$:

$T_0 = R$

I.S. $n \mapsto n + 1$:

$T_{n+1} = R \cup T_n \circ R.$

The relation T_n can be expressed via the relation R, we have

1. $T_0 = R$.
2. $T_1 = R \cup T_0 \circ R = R \cup R \circ R = R^1 \cup R^2$.
3. $$
   \begin{aligned}
   T_2 &= R \cup T_1 \circ R \\
   &= R \cup (R^1 \cup R^2) \circ R \\
   &= R^1 \cup R^2 \cup R^3.
   \end{aligned}
   $$

In general, we can show by induction that

$$T_n = \bigcup_{i=1}^{n+1} R^i$$

holds for all $n \in \mathbb{N}$. The base case of this proof is immediate from the definition of T_0. In the induction step we observe the following:

$$
\begin{aligned}
T_{n+1} &= R \cup T_n \circ R && \text{(by definition)} \\
&= R \cup \left(\bigcup_{i=1}^{n+1} R^i \right) \circ R && \text{(by induction hypothesis)} \\
&= R \cup \left(R \cup \cdots \cup R^{n+1} \right) \circ R \\
&= R^1 \cup R^2 \cup \cdots \cup R^{n+2} && \text{(by the distributivity of } \circ \text{ over } \cup) \\
&= \bigcup_{i=1}^{n+2} R^i && \square
\end{aligned}
$$

The sequence $(T_n)_{n \in \mathbb{N}}$ has another useful property: It is monotonically increasing. In general, a sequence of sets $(X_n)_{n \in \mathbb{N}}$ is called monotonically increasing iff we have

$$\forall n \in \mathbb{N} : X_n \subseteq X_{n+1},$$

i.e. the sets X_n get bigger with growing index n. The monotonicity of the sequence $(T_n)_{n \in \mathbb{N}}$ is an immediate

consequence of the equation

$$T_n = \bigcup_{i=1}^{n+1} R^i$$

because we have:

$$\begin{aligned} & T_n \subseteq T_{n+1} \\ \Leftrightarrow \quad & \bigcup_{i=1}^{n+1} R^i \subseteq \bigcup_{i=1}^{n+2} R^i \\ \Leftrightarrow \quad & \bigcup_{i=1}^{n+1} R^i \subseteq \bigcup_{i=1}^{n+1} R^i \cup R^{n+2} \end{aligned}$$

If the relation R is finite, then the transitive closure R^+ is finite, too. The sets T_n are all subsets of R^+ because we have

$$T_n = \bigcup_{i=1}^{n+1} R^i \subseteq \bigcup_{i=1}^{\infty} R^i = R^+ \quad \text{for all } n \in \mathbb{N}.$$

Hence the sets T_n can not grow indefinitely. Because of the monotonicity of the sequence $(T_n)_{n\in\mathbb{N}}$ it follows that there exists an index $k \in \mathbb{N}$ such that the sets T_n do not grow any further once n has reached k, i.e. we have

$$\forall n \in \mathbb{N} : (n \geq k \rightarrow T_n = T_k).$$

But this implies that

$$T_n = \bigcup_{i=1}^{n+1} R^i = \bigcup_{i=1}^{\infty} R^i = R^+ \quad \text{holds for all } n \geq k.$$

Therefore, the algorithm for computing R^+ iterates the equation

$$T_{n+1} = R \cup T_n \circ R$$

until the equation $T_{n+1} = T_n$ is satisfied, since this implies that $T_n = R^+$.

The program `transitive-closure.py` that is shown in Figure 4.6 on page 45 shows an implementation of this idea. The program produces the following output:

```
R  = {(1, 2), (1, 3), (4, 5), (2, 3), (2, 4)}
Computing the transitive closure of R:
R+ = {(1, 2), (1, 3), (4, 5), (1, 4), (1, 5), (2, 3), (2, 5), (2, 4)}
```

The transitive closure R^+ of a relation R has a very intuitive interpretation: It contains all pairs $\langle x, y\rangle$ such that there is a path leading from x to y. The function $\texttt{product}(R_1, R_2)$ computes the relational product $R_1 \circ R_2$ according to the formula

$$R_1 \circ R_2 = \{\langle x, z\rangle \mid \exists y : \big(\langle x, y\rangle \in R_1 \wedge \langle y, z\rangle \in R_2\big)\}.$$

4.3.2 Computing the Paths

So far, given a graph represented by a relation R and two vertices x and y, we can only check whether there is a path leading from x to y, but we cannot compute this path. In this subsection we will extend the procedure `transClosure` so that it will also compute the corresponding path. The main idea is to extend the notion of a relational product to the notion of a path product, where a path product is defined on sets of paths. In order to do so, we introduce three functions for tuples.

1. Given a tuple T, the function $\texttt{first}(T)$ returns the first element of T:

```
1  def product(R1, R2):
2      "Compute the relational product of R1 and R2."
3      return { (x, z) for (x, y1) in R1 for (y2, z) in R2 if y1 == y2 }
4
5  def transClosure(R):
6      "Compute the transitive closure of the binary relation R."
7      T = R
8      while True:
9          oldT = T
10         T    = product(R,T).union(R)
11         if T == oldT:
12             return T
13
14 R = { (1,2), (2,3), (1,3), (2,4), (4,5) }
15 print( "R  = ", R );
16 print( "Computing the transitive closure of R:" );
17 T = transClosure(R);
18 print( "R+ = ", T );
```

Abbildung 4.6: Computing the transitive closure.

$$\texttt{first}\big(\langle x_1, \cdots, x_m\rangle\big) = x_1.$$

2. Given a tuple T, the function $\texttt{last}(T)$ returns the last element of T:

$$\texttt{last}\big(\langle x_1, \cdots, x_m\rangle\big) = x_m.$$

3. If $S = \langle x_1, \cdots, x_m\rangle$ and $T = \langle y_1, \cdots, y_n\rangle$ are two tuples such that $\texttt{first}(S) = \texttt{last}(S)$, we define the join $S \oplus T$ of S and T as

$$S \oplus T = \langle x_1, \cdots, x_m, y_2, \cdots, y_n\rangle.$$

If $\mathcal{P}_1$ and $\mathcal{P}_2$ are sets of tuples representing paths, we define the path product of $\mathcal{P}_1$ and $\mathcal{P}_2$ as follows:

$$\mathcal{P}_1 \bullet \mathcal{P}_2 = \big\{\, T_1 \oplus T_2 \mid T_1 \in \mathcal{P}_1 \wedge T_2 \in \mathcal{P}_2 \wedge \texttt{last}(T_1) = \texttt{first}(T_2) \,\big\}.$$

Using the notion of a path product we are able to extend the program shown in Figure 4.6 such that it computes all paths between two vertices. The resulting program `path.py` is shown in Figure 4.7 on page 46. Unfortunately, the program does not work any more if the graph is cyclic. A graph is defined to be cyclic if there is a path of length greater than 1 that starts and ends at the same vertex. This path is then called a cycle. Figure 4.8 on page 46 shows a cyclic graph. This graph is cyclic because it contains the path

$$\langle 1, 2, 4, 1\rangle$$

and this path is a cycle. The problem with this graph is that it contains an infinite number of paths that connect the vertex 1 with the vertex 2:

$$\langle 1, 2\rangle, \langle 1, 2, 4, 1, 2\rangle, \langle 1, 2, 4, 1, 2, 4, 1, 2\rangle, \langle 1, 2, 4, 1, 2, 4, 1, 2, 4, 1, 2\rangle, \cdots$$

Of course, there is no point in computing a path that visits a vertex more than once as these paths contain cycles. Our goal is to eliminate all those paths that contain cycles.

Figure 4.9 on page shows how the implementation of the function `pathProduct` has to be changed so that the resulting program `path-cyclic.py` works also for cyclic graphs.

1. In line 2 and 3, we compute only those paths that are not cyclic.
2. Line 6 defines a function `noCycle` that tests, whether the join `T1` $\oplus$ `T2` is cyclic. The join of `T1` and `T2` is cyclic iff the tuples `T1` and `T2` have more than one common element. The tuples `T1` and `T2` will always

```
def findPaths(R):
    P = R;
    while True:
        oldP = P
        P    = R.union(pathProduct(P, R))
        if P == oldP:
            return P

def pathProduct(P, Q):
    return { join(S, T) for S in P for T in Q if S[-1] == T[0] }

def join(S, T):
    return S + T[1:]

R = { (1,2), (2,3), (1,3), (2,4), (4,5) }
print("R = ", R)
print("Computing all paths:" )
P = findPaths(R)
print("P = ", P)
```

Abbildung 4.7: Computing all connections.

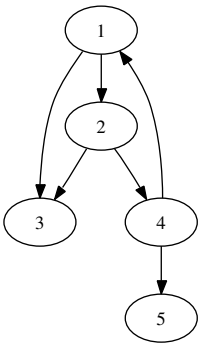

Abbildung 4.8: A graph with a cycle.

```
def pathProduct(P, Q):
    return { join(S, T) for S in P for T in Q
                        if S[-1] == T[0] and noCycle(S, T)
           }

def noCycle(T1, T2):
    return len({ x for x in T1 }.intersection({ x for x in T2 })) == 1
```

Abbildung 4.9: Computing the connections in a cyclic graph.

have at least one common element, as we join these tuples only if the last element of `T1` is equal to the first element of `T2`. If there would be an another vertex common to both `T1` and `T2`, then the path `T1` $\oplus$ `T2` would be cyclic.

In general, we are not really interested to compute all possible paths between two given vertices `x` and

y. Instead, we just want to compute the shortest path leading from x to y. Figure 4.10 on page 48 shows the procedure reachable. This procedure takes three arguments:

1. start and goal are vertices of a graph.
2. R is a binary relation representing a directed graph.

The call reachable(start, goal, R) checks whether start and goal are connected and, furthermore, computes the shortest path from start to goal, provided such a path exists. The complete program can be found in the file find_path.py. Next, we discuss the implementation of the procedure reachable.

1. Line 2 initializes the set P. After n iterations, this set will contain all paths that start with the vertex start and that have a length of at most n.

   Initially, there is just the trivial path $\langle$start$\rangle$ that starts with vertex start and has length 0.
2. Line 5 tries to extend all previously computed paths by one step.
3. Line 6 selects all those paths from the set P that lead to the vertex goal. These paths are stored in the set Found.
4. Line 7 checks whether we have indeed found a path ending at goal. This is the case if the set Found is not empty. In this case, we return any of these paths.
5. If we have not yet found the vertex goal and, furthermore, we have not been able to find any new paths during this iteration, the procedure returns in line 10. As the return statement in line 11 does not return a value, the procedure will instead return the value None.

The procedure call reachable(start,goal R) will compute the **shortest** path connecting start and goal because it computes path with increasing length. The first iteration computes all paths starting in start that have a length of at most 1, the second iteration computes all paths starting in start that have a length of at most 2, and in general the n-th iteration computes all paths starting in start that have a length of at most n. Hence, if there is a path of length n, then this path will be found in the n-iteration unless a shorter path has already been found in a previous iteration.

Remark: The algorithm described above is known as breadth first search. ◇

4.3.3 The Wolf, the Goat, and the Cabbage

Next, we present an application of the theory developed so far. We solve a problem that has puzzled the greatest agricultural economists for centuries. The puzzle we want to solve is known as the wolf-goat-cabbage puzzle:

> *An agricultural economist has to sell a wolf, a goat, and a cabbage on a market place. In order to reach the market place, she has to cross a river. The boat that she can use is so small that it can only accommodate either the goat, the wolf, or the cabbage in addition to the agricultural economist. Now if the agricultural economist leaves the wolf alone with the goat, the wolf will eat the goat. If, instead, the agricultural economist leaves the goat with the cabbage, the goat will eat the cabbage. Is it possible for the agricultural economist to develop a schedule that allows her to cross the river without either the goat or the cabbage being eaten?*

In order to compute a schedule, we first have to model the problem. The various states of the problem will be regarded as vertices of a graph and this graph will be represented as a binary relation. To this end we define the set

```
All = {'farmer', 'wolf, 'goat', 'cabbage'}
```

```
def reachable(start, goal, R):
    P = { (start,) }
    while True:
        oldP  = P
        P     = P.union(path_product(P, R))
        Found = { T for T in P if T[-1] == goal }
        if Found != set():
            return Found.pop()
        if P == oldP:
            return

def path_product(P, R):
    return set( add(T1, T2) for T1 in P for T2 in R
                            if T1[-1] == T2[0] and noCycle(T1, T2)
              )

def noCycle(T1, T2):
    return len(set(T1).intersection(set(T2))) == 1

def add(T, P):
    return T + (P[-1],)
```

Abbildung 4.10: Finding the shortest path between two vertices.

Every node will be represented as a subset `S` of the set `All`. The idea is that the set `S` specifies those objects that are on the left side of the river. We assume that initially the farmer and his goods are on the left side of the river. Therefore, the set of all possible states can be defined as the set

```
States = {S for S in power(All) if not problem(S) and not problem(All-S)}
```

Here, we have used the procedure `problem` to check whether a given set `S` has a problem. Note that since `S` is the set of objects on the left side, the expression `All-S` computes the set of objects on the right side of the river.

Next, a set `S` of objects has a problem if both of the following conditions are satisfied:

1. The farmer is not an element of `S` and
2. either `S` contains both the goat and the cabbage or `S` contains both the wolf and the goat.

Therefore, we can implement the function `problem` as follows:

```
def problem(S):
    return ("farmer" not in S) and             \
           (("goat" in S and "cabbage" in S) or   # goat eats cabbage
            ("wolf" in S and "goat"    in S)   )  # wolf eats goat
```

We proceed to compute the relation `R` that contains all possible transitions between different states. We will compute `R` using the formula:

```
R = R1 + R2;
```

Here `R1` describes the transitions that result from the farmer crossing the river from left to right, while `R2` describes the transitions that result from the farmer crossing the river from right to left. We can define the relation `R1` as follows:

```
R1 = { (S, S-B) for S in States for B in power(S)
                if S-B in States and 'farmer' in B and len(B) <= 2
     }
```

Let us explain this definition in detail:

1. Initially, `S` is the set of objects on the left side of the river. Hence, `S` is an element of the set of all states that we have defined as `P`.
2. `B` is the set of objects that are put into the boat and that do cross the river. Of course, for an object to go into the boat is has to be on the left side of the river to begin with. Therefore, `B` is a subset of `S` and hence an element of the power set of `S`.
3. Then `S-B` is the set of objects that are left on the left side of the river after the boat has crossed. Of course, the new state `S-B` has to be a state that does not have a problem. Therefore, we check that `S-B` is an element of `States`.
4. Furthermore, the farmer has to be inside the boat. This explains the condition

   ```
   'farmer' in B.
   ```

5. Finally, the boat can only have two passengers. Therefore, we have added the condition

   ```
   len(B) <= 2.
   ```

Next, we have to define the relation `R2`. However, as crossing the river from right to left is just the reverse of crossing the river from left to right, `R2` is just the inverse of `R1`. Hence we define:

```
R2 = { (S2, S1) for (S1, S2) in R1 }
```

Next, the relation `R` is the union of `R1` and `R2`:

```
R = R1.union(R2)
```

Finally, the start state has all objects on the left side. Therefore, we have

```
start = All
```

In the end, all objects have to be on the right side of the river. That means that nothing is left on the left side. Therefore, we define

```
goal = {}
```

Figure 4.12 on page 50 shows the program `wolf-goat-cabbage.py` that combines the statements shown so far. The solution computed by this program is shown in Figure 4.13.

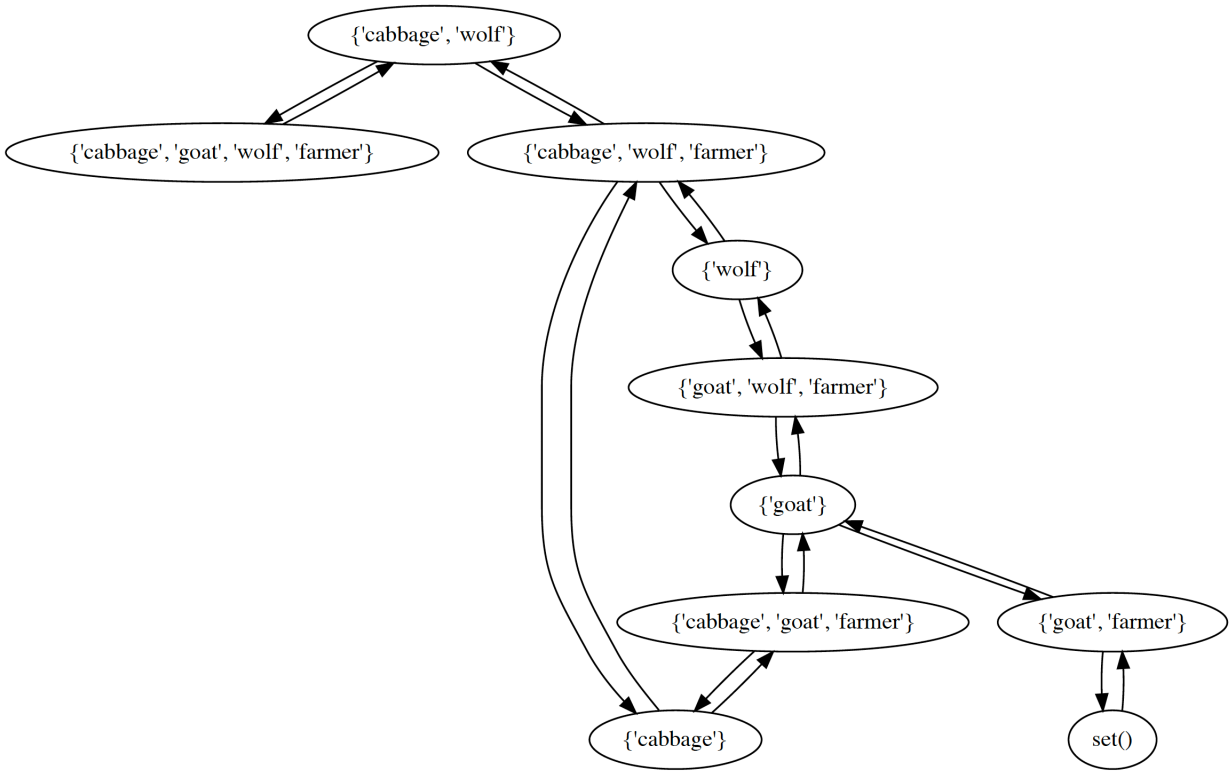

Abbildung 4.11: The relation `R` shown as a directed graph.

```
def problem(S):
    return ('farmer' not in S) and             \
           (('goat' in S and 'cabbage' in S) or   # goat eats cabbage
            ('wolf' in S and 'goat'    in S)   )  # wolf eats goat

All   = frozenset( {'farmer', 'wolf', 'goat', 'cabbage'} )
R1    = { (S, S - B) for S in States for B in power(S)
                     if S - B in States and 'farmer' in B and len(B) <= 2
        }
R2    = { (S2, S1) for (S1, S2) in R1 }
R     = R1.union(R2)
start = All
goal  = frozenset()
Path  = findPath(start, goal, R)
```

Abbildung 4.12: Solving the wolf-goat-cabbage problem.

```
{"cabbage", "farmer", "goat", "wolf"}                                  {}
                          >>>> {"farmer", "goat"} >>>>
{"cabbage", "wolf"}                                    {"farmer", "goat"}
                          <<<< {"farmer"} <<<<
{"cabbage", "farmer", "wolf"}                                        {"goat"}
                          >>>> {"farmer", "wolf"} >>>>
{"cabbage"}                                      {"farmer", "goat", "wolf"}
                          <<<< {"farmer", "goat"} <<<<
{"cabbage", "farmer", "goat"}                                        {"wolf"}
                          >>>> {"cabbage", "farmer"} >>>>
{"goat"}                                        {"cabbage", "farmer", "wolf"}
                          <<<< {"farmer"} <<<<
{"farmer", "goat"}                                     {"cabbage", "wolf"}
                          >>>> {"farmer", "goat"} >>>>
{}                                    {"cabbage", "farmer", "goat", "wolf"}
```

Abbildung 4.13: A schedule for the agricultural economist.

4.4 Symbolic Differentiation

In this section we will develop a program that reads an arithmetic expression like

```
"x * exp(x)",
```

interprets this string as describing the real valued function

$$x \mapsto x \cdot \exp(x),$$

and then takes the derivative of this function with respect to the variable x. In order to specify the input of this program more clearly, we first define the notion of an arithmetic expression inductively.

1. Every number $c \in \mathbb{R}$ is an arithmetic expression.
2. Every variable v is an arithmetic expression.
3. If s and t are arithmetic expressions, then
   $$s + t, \quad s - t, \quad s * t, \quad s/t, \quad \text{and} \quad s ** t$$
   are arithmetic expressions. Here $s ** t$ is interpreted as s^t.
4. If e is an arithmetic expression, then both
   $$\exp(e) \quad \text{and} \quad \ln(e)$$
   are arithmetic expressions.

We want do implement a function `diff` that takes two arguments:

1. The first argument `expr` is an arithmetic expression.
2. The second argument `var` is the name of a variable.

The function call `diff(expr, var)` will then compute the derivative of `expr` with respect to the variable `var`. For example, the function call `diff("x*exp(x)", "x")` will compute the output

```
"1*exp(x) + x*exp(x)"
```

because we have:

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(x \cdot \mathrm{e}^x\right) = 1 \cdot x + x \cdot \mathrm{e}^x$$

It would be very tedious to represent arithmetic expressions as strings. Instead, we will represent arithmetic expressions as nested tuples. The notion of a *nested tuple* is defined inductively:

- $\langle x_1, x_2, \cdots, x_n \rangle$ is a nested tuple if each of the components x_i is either a number, a string, or is itself a nested tuple.

For example, the arithmetic expression "`x*exp(x)`" is represented as the nested tuple

$$\big\langle \texttt{"*"}, \texttt{"}x\texttt{"}, \big\langle \texttt{"exp"}, \texttt{"}x\texttt{"} \big\rangle \big\rangle.$$

In order to be able to convert string into nested tuples, we need a parser. A parser is a program that takes a string as input and transforms this string into a nested tuple, which is then returned as a result. I have implemented a parser in the file "`exprParser.py`". The details of the implementation of this parser will be discussed in the lecture on algorithms.

The function `diff` that is shown in Figure 4.14 on page 54 is part of the program `diff.py`. This function is called with one argument: The argument `e` is an arithmetic expression. The function `diff` interprets its argument `e` as a function of the variable `x`. We take the derivative of this function with respect to the variable `x`. For example, in order to compute the derivative of the function

$$x \mapsto x^x,$$

we can call the function `diff` as follows:

```
diff("x ** x").
```

Let us now discuss the implementation of the function `diff` in more detail.

1. The lines 3 - 6 implement the rule:

$$\frac{\mathrm{d}}{\mathrm{d}x}\big(f(x) + g(x)\big) = \frac{\mathrm{d}}{\mathrm{d}x}f(x) + \frac{\mathrm{d}}{\mathrm{d}x}g(x)$$

2. Line 7 - 10 implement the rule:

$$\frac{\mathrm{d}}{\mathrm{d}x}\big(f(x) - g(x)\big) = \frac{\mathrm{d}}{\mathrm{d}x}f(x) - \frac{\mathrm{d}}{\mathrm{d}x}g(x)$$

3. Line 11 - 14 deals with the case where `e` is a product. The product rule is

$$\frac{\mathrm{d}}{\mathrm{d}x}\big(f(x) \cdot g(x)\big) = \left(\frac{\mathrm{d}}{\mathrm{d}x}f(x)\right) \cdot g(x) + f(x) \cdot \left(\frac{\mathrm{d}}{\mathrm{d}x}g(x)\right)$$

4. Line 15 - 17 deals with the case where `e` is a quotient. The quotient rule is

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{f(x)}{g(x)}\right) = \frac{\left(\frac{\mathrm{d}}{\mathrm{d}x}f(x)\right) \cdot g(x) - f(x) \cdot \left(\frac{\mathrm{d}}{\mathrm{d}x}g(x)\right)}{g(x) \cdot g(x)}$$

5. Line 19 - 21 deals with the case where `e` is a power. Now in order to take the derivative of an expression of the form

$$f(x)^{g(x)}$$

we first need to rewrite this expression using the following trick:

$$f(x)^{g(x)} = \exp\big(\ln\big(f(x)^{g(x)}\big)\big) = \exp\big(g(x) \cdot \ln(f(x))\big)$$

Then, we can recursively call `diff` for this expression. This works, because the function `diff` can deal with both the exponential function $x \mapsto \exp(x)$ and with the natural logarithm $x \mapsto \ln(x)$. This rewriting is done in line 21.

6. Line 22-25 deals with the case where `e` has the form

$$\ln\big(f(x)\big)$$

In order to take the derivative of this expression, we first need to know the derivative of the natural logarithm. This derivative is given as

$$\frac{\mathrm{d}}{\mathrm{d}x}\ln(x) = \frac{1}{x}$$

Then, using the chain rule we have that

$$\frac{\mathrm{d}}{\mathrm{d}x}\ln\big(f(x)\big) = \frac{\frac{\mathrm{d}}{\mathrm{d}x}f(x)}{f(x)}$$

7. Line 26 - 29 deals with the case where `e` has the form $\exp\big(f(x)\big)$. In order to take the derivative of this expression, we first need to know the derivative of the exponential function. This derivative is given as

$$\frac{\mathrm{d}}{\mathrm{d}x}\exp(x) = \exp(x)$$

Then, using the chain rule we have that

$$\frac{\mathrm{d}}{\mathrm{d}x}\exp\big(f(x)\big) = \left(\frac{\mathrm{d}}{\mathrm{d}x}f(x)\right)\cdot\exp\big(f(x)\big)$$

8. Line 30-31 deals with the case where `e` is a variable and happens to be the same variable as `x`. This is checked using the condition `e == x`. As we have

$$\frac{\mathrm{d}x}{\mathrm{d}x} = 1,$$

the function `diff` returns `1` in this case.

9. Otherwise, the expression is assumed to be a constant and hence we return 0.

In order to test this function we can implement a function `test` as shown in Figure 4.15. Then the expression

```
diff("x ** x")
```

yields the result:

```
d/dx x ** x = (1*ln(x) + x*1/x)*exp(x*ln(x))
```

This shows that

$$\frac{\mathrm{d}}{\mathrm{d}x}x^x = \big(\ln(x)+1\big)\cdot\exp\big(x\cdot\ln(x)\big) = \big(\ln(x)+1\big)\cdot x^x.$$

```
def diff(e):
    "differentiate the expressions e with respect to the variable x"
    if e[0] == '+':
        f , g  = e[1:]
        fs, gs = diff(f), diff(g)
        return ('+', fs, gs)
    if e[0] == '-':
        f , g  = e[1:]
        fs, gs = diff(f), diff(g)
        return ('-', fs, gs)
    if e[0] == '*':
        f , g  = e[1:]
        fs, gs = diff(f), diff(g)
        return ('+', ('*', fs, g), ('*', f, gs))
    if e[0] == '/':
        f , g  = e[1:]
        fs, gs = diff(f), diff(g)
        return ('/', ('-', ('*', fs, g), ('*', f, gs)), ('*', g, g))
    if e[0] == '**':
        f , g  = e[1:]
        return diff(('exp', ('*', g, ('ln', f))))
    if e[0] == 'ln':
        f  = e[1]
        fs = diff(f)
        return ('/', fs, f)
    if e[0] == 'exp':
        f  = e[1]
        fs = diff(f)
        return ('*', fs, e)
    if e == 'x':
        return '1'
    return 0
```

Abbildung 4.14: A function for symbolic differentiation

```
import exprParser as ep

def test(s):
    t = ep.ExprParser(s).parse()
    d = diff(t)
    print(f"d/dx {s} = {ep.toString(d)}")
```

Abbildung 4.15: Testing symbolic differentiation.

Kapitel 5

Grenzen der Berechenbarkeit

In jeder Disziplin der Wissenschaft wird die Frage gestellt, welche Grenzen die verwendeten Methoden haben. Wir wollen daher in diesem Kapitel beispielhaft ein Problem untersuchen, bei dem die Informatik an ihre Grenzen stößt. Es handelt sich um das Halte-Problem.

5.1 Das Halte-Problem

Das Halte-Problem ist die Frage, ob eine gegebene Funktion f für eine bestimmte Eingabe x terminiert, ob also der Aufruf $f(x)$ ein Ergebnis liefert oder sich in eine Endlos-Schleife verabschiedet. Bevor wir formal beweisen, dass das Halte-Problem im Allgemeinen unlösbar ist, wollen wir versuchen, anschaulich zu verstehen, warum dieses Problem schwer sein muss. Dieser informalen Betrachtung des Halte-Problems ist der nächste Abschnitt gewidmet. Im Anschluss an diesen Abschluss beweisen wir dann formal die Unlösbarkeit des Halte-Problems.

5.1.1 Informale Betrachtungen zum Halte-Problem

Um zu verstehen, warum das Halte-Problem schwer ist, betrachten wir das in Abbildung 5.1 gezeigte Programm. Dieses Programm ist dazu gedacht, die *Legendresche Vermutung* zu überprüfen. Der französische Mathematiker Adrien-Marie Legendre (1752 — 1833) hatte vor etwa 200 Jahren die Vermutung aufgestellt, dass zwischen zwei Quadrat-Zahlen immer eine Primzahl liegt. Die Frage, ob diese Vermutung richtig ist, ist auch heute noch unbeantwortet. Die in Abbildung 5.1 definierte Funktion `legendre`(n) überprüft für eine gegebene positive natürliche Zahl n, ob zwischen n^2 und $(n+1)^2$ eine Primzahl liegt. Falls dies, wie von Legendre vermutet, der Fall ist, gibt die Funktion als Ergebnis `True` zurück, andernfalls wird `False` zurück gegeben. Die Funktion `legendre` ist mit Hilfe der Funktion `is_prime` definiert. Für eine natürliche Zahl k liefert `is_prime`(k) genau dann den Wert `True` zurück, wenn k eine Primzahl ist. Dazu überprüft die Funktion `is_prime`(k) ob die Menge der Teiler von k mit der Menge die Menge $\{1, k\}$ übereinstimmt. Die Menge der Teiler wird mit Hilfe der Funktion `divisors` berechnet.

Abbildung 5.1 enthält darüber hinaus die Definition der Funktion `findCounterExample`(n), die versucht, für eine gegebene positive natürliche Zahl n eine Zahl $k \geq n$ zu finden, so dass zwischen k^2 und $(k+1)^2$ keine Primzahl liegt. Die Idee bei der Implementierung dieser Funktion ist einfach: Zunächst überprüfen wir durch den Aufruf `legendre`(n), ob zwischen n^2 und $(n+1)^2$ eine Primzahl liegt. Falls dies der Fall ist, untersuchen wir anschließend das Intervall von $(n+1)^2$ bis $(n+2)^2$, dann das Intervall von $(n+2)^2$ bis $(n+3)^2$ und so weiter, bis wir schließlich eine Zahl m finden, so dass zwischen m^2 und $(m+1)^2$ keine Primzahl liegt. Falls Legendre Recht hatte, werden wir nie ein solches k finden und in diesem Fall wird der Aufruf `findCounterExample`(2) nicht terminieren.

Nehmen wir nun an, wir hätten ein schlaues Programm, nennen wir es `stops`, das als Eingabe eine *Python* Funktion f und ein Argument a verarbeitet und das uns die Frage, ob die Berechnung von $f(a)$ terminiert, beantworten kann. Die Idee wäre, dass die Funktion `stops` die folgende Spezifikation erfüllt:

```
def legendre(n):
    k = n * n + 1;
    while k < (n + 1) ** 2:
        if is_prime(k):
            return True
        k += 1
    return False

def is_prime(k):
    return divisors(k) == {1, k}

def divisors(k):
    return { t for t in range(1, k+1) if k % t == 0 }

def find_counter_example(n):
    while True:
       if legendre(n):
           n = n + 1
       else:
           print(f'Eureka! No prime between {n}**2 and {n+1}**2!')
           return
```

Abbildung 5.1: Eine Funktion zur Überprüfung der von Legendre aufgestellten Vermutung.

$\texttt{stops}(f, a) = \texttt{true}$ g.d.w. der Aufruf $f(a)$ terminiert.

Falls der Aufruf $f(a)$ nicht terminiert, sollte stattdessen $\texttt{stops}(f, a) = \texttt{false}$ gelten. Wenn wir eine solche Funktion `stops` hätten, dann könnten wir

```
stops(findCounterExample, 1)
```

aufrufen und wüssten anschließend, ob die Vermutung von Legendre wahr ist oder nicht: Wenn

```
stops(findCounterExample, 1) = true
```

ist, dann würde das heißen, dass der Funktions-Aufruf `findCounterExample`(1) terminiert. Das passiert aber nur dann, wenn ein Gegenbeispiel gefunden wird. Würde der Aufruf

```
stops(findCounterExample, 1)
```

stattdessen den Wert `false` zurück liefern, so könnten wir schließen, dass der Aufruf `findCounterExample(1)` nicht terminiert. Mithin würde die Funktion `findCounterExample` kein Gegenbeispiel finden und damit wäre klar, dass die von Legendre aufgestellte Vermutung wahr ist.

Es gibt eine Reihe weiterer offener mathematischer Probleme, die alle auf die Frage abgebildet werden können, ob eine gegebene Funktion terminiert. Daher zeigen die vorhergehenden Überlegungen, dass es sehr nützlich wäre, eine Funktion wie `stops` zur Verfügung zu haben. Andererseits können wir an dieser Stelle schon ahnen, dass die Implementierung der Funktion `stops` nicht ganz einfach sein kann.

5.1.2 Formale Analyse des Halte-Problems

Wir werden in diesem Abschnitt beweisen, dass das Halte-Problem unlösbar ist. Dazu führen wir den Begriff einer Test-Funktion ein.

Definition 1 (Test-Funktion) *Ein String t ist genau dann eine Test-Funktion mit Namen f, wenn t die Form*

```
"""
def f(x):
   body
"""
```

hat und sich als Python-Funktion parsen lässt. Die Variable body steht hier für den Rumpf der Test-Funktion. Die Menge der Test-Funktionen bezeichnen wir mit TF. ◇

Beispiele:

1. s_1 :=
```
"""
def zero(x):
    return 0
"""
```
   s_1 ist eine (sehr einfache) Test-Funktion mit dem Name `zero`.

2. s_2 :=
```
"""
def loop(x):
    while True:
        x = x + 1
"""
```
   s_2 ist eine Test-Funktion mit dem Name `loop`.

3. s_3 :=
```
"""
def buggy(x):
    while True:
        ++x
"""
```
   s_3 ist keine Test-Funktion, denn da *Python* den Präfix-Operator "++" nicht unterstützt, lässt sich der String s_3 nicht fehlerfrei als *Python* parsen.

4. s_4 :=
```
"""
def sum(x, y):
    while True:
        return x + y
"""
```
   s_4 ist keine Test-Funktion, denn ein String ist nur dann eine Test-Funktion, wenn die durch den String definierte Funktion mit genau einen Parameter aufgerufen wird.

Um das Halte-Problem übersichtlicher formulieren zu können, führen wir noch vier zusätzliche Notationen ein.

Notation 2 (*name*, $\rightsquigarrow$, $\downarrow$, $\uparrow$) Ist t eine Testfunktion mit Namen f, so schreiben wir

$$name(t) = f.$$

Ist f der Name einer *Python*-Funktion, die k Argumente verarbeitet und sind $a_1, \cdots, a_k$ mögliche Argumente, mit denen wir diese Funktion aufrufen können, so schreiben wir

$$f(a_1, \cdots, a_k) \rightsquigarrow r$$

wenn der Aufruf $f(a_1, \cdots, a_k)$ das Ergebnis r liefert. Sind wir an dem Ergebnis selbst nicht interessiert, sondern wollen nur angeben, dass ein Ergebnis existiert, so schreiben wir

$$f(a_1, \cdots, a_k) \downarrow$$

und sagen, dass der Aufruf $f(a_1, \cdots, a_k)$ terminiert. Terminiert der Aufruf $f(a_1, \cdots, a_k)$ nicht, so schreiben wir

$$f(a_1, \cdots, a_k) \uparrow$$

und sagen, dass der Aufruf $f(a_1, \cdots, a_k)$ divergiert. Diese Notation verwenden wir auch, wenn der Aufruf $f(a_1, \cdots, a_k)$ mit einer Fehlermeldung abbricht. □

Beispiele: Legen wir die Funktions-Definitionen zugrunde, die wir im Anschluss an die Definition des Begriffs der Test-Funktion gegeben haben, so gilt:

1. `zero(1)` $\rightsquigarrow 0$
2. `zero(1)` $\downarrow$
3. `loop(0)` $\uparrow$

Das Halte-Problem für *Python*-Funktionen ist die Frage, ob es eine *Python*-Funktion

```
def stops(t, a):
    body
```

gibt, die als Eingaben eine Testfunktion t und einen String a erhält und die folgende Eigenschaft hat:

1. $t \notin TF \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 2.$

   Der Aufruf `stops(t, a)` liefert genau dann den Wert 2 zurück, wenn t keine Test-Funktion ist.

2. $t \in TF \wedge name(t) = f \wedge f(a) \downarrow \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 1.$

   Der Aufruf `stops(t, a)` liefert genau dann den Wert 1 zurück, wenn t eine Test-Funktion mit Namen f ist und der Aufruf $f(a)$ terminiert.

3. $t \in TF \wedge name(t) = f \wedge t(a) \uparrow \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 0.$

   Der Aufruf `stops(t, a)` liefert genau dann den Wert 0 zurück, wenn t eine Test-Funktion ist und der Aufruf $t(a)$ nicht terminiert.

Falls eine *Python*-Funktion `stops` mit den obigen Eigenschaften existiert, dann sagen wir, dass das Halte-Problem für *Python* entscheidbar ist.

Theorem 3 (Alan Turing, 1936) Das Halte-Problem ist unentscheidbar.

Beweis: Zunächst eine Vorbemerkung. Um die Unentscheidbarkeit des Halte-Problems nachzuweisen, müssen wir zeigen, dass etwas, nämlich eine Funktion mit gewissen Eigenschaften nicht existiert. Wie kann so ein Beweis überhaupt funktionieren? Wie können wir überhaupt zeigen, dass irgendetwas nicht existiert? Die einzige Möglichkeit zu zeigen, dass etwas nicht existiert ist indirekt: Wir nehmen also an, dass eine Funktion `stops` existiert, die das Halte-Problem löst. Aus dieser Annahme werden wir einen Widerspruch ableiten. Dieser Widerspruch zeigt uns dann, dass eine Funktion `stops` mit den gewünschten Eigenschaften nicht existieren kann. Um zu einem Widerspruch zu kommen, definieren wir den String `turing` wie in Abbildung 5.2 gezeigt.

Mit dieser Definition ist klar, dass `turing` eine Test-Funktion mit Namen `alan` ist:

$$\texttt{turing} \in TF \quad \text{und} \quad name(\texttt{turing}) = \texttt{alan}.$$

Damit sind wir in der Lage, den String `turing` als Eingabe der Funktion `stops` zu verwenden. Wir betrachten nun den folgenden Aufruf:

```
stops(turing, turing);
```

Da `turing` eine Test-Funktion mit dem Namen `alan` ist und der Aufruf von `alan` mit dem Argument `turing`

```
1  turing = """
2            def alan(x):
3                result = stops(x, x)
4                if result == 1:
5                    while True:
6                        print("... looping ...")
7                return result
8            """
```

Abbildung 5.2: Die Definition des Strings `turing`.

auch nicht zu einem Fehler führen darf, können nur zwei Fälle auftreten:

$$\texttt{stops(turing, turing)} \rightsquigarrow 0 \quad \vee \quad \texttt{stops(turing, turing)} \rightsquigarrow 1.$$

Diese beiden Fälle analysieren wir nun im Detail:

1. $\texttt{stops(turing, turing)} \rightsquigarrow 0$.

   Nach der Spezifikation von `stops` bedeutet dies

   $$\texttt{alan(turing)} \uparrow$$

   Schauen wir nun, was wirklich beim Aufruf `alan(turing)` passiert: In Zeile 2 erhält die Variable `result` den Wert 0 zugewiesen. In Zeile 3 wird dann getestet, ob `result` den Wert 1 hat. Dieser Test schlägt fehl. Daher wird der Block der `if`-Anweisung nicht ausgeführt und die Funktion liefert als nächstes in Zeile 8 den Wert 0 zurück. Insbesondere terminiert der Aufruf also, im Widerspruch zu dem, was die Funktion `stops` behauptet hat. ↯

   Damit ist der erste Fall ausgeschlossen.

2. $\texttt{stops(turing, turing)} \rightsquigarrow 1$.

   Aus der Spezifikation der Funktion `stops` folgt, dass der Aufruf `alan(turing)` terminiert:

   $$\texttt{alan(turing)} \downarrow$$

   Schauen wir nun, was wirklich beim Aufruf `alan(turing)` passiert: In Zeile 2 erhält die Variable `result` den Wert 1 zugewiesen. In Zeile 3 wird dann getestet, ob `result` den Wert 1 hat. Diesmal gelingt der Test. Daher wird der Block der `if`-Anweisung ausgeführt. Dieser Block besteht aber nur aus einer Endlos-Schleife, aus der wir nie wieder zurück kommen. Das steht im Widerspruch zu dem, was die Funktion `stops` behauptet hat. ↯

   Damit ist der zweite Fall ausgeschlossen.

Insgesamt haben wir also in jedem Fall einen Widerspruch erhalten. Damit muss die Annahme, dass die *Python*-Funktion `stops` das Halte-Problem löst, falsch sein, denn diese Annahme ist die Ursache für die Widersprüche, die wir erhalten haben. Insgesamt haben wir daher gezeigt, dass es keine *Python*-Funktion geben kann, die das Halte-Problem löst. □

Bemerkung: Der Nachweis, dass das Halte-Problem unlösbar ist, wurde 1936 von Alan Turing (1912 – 1954) [Tur36] erbracht. Turing hat das Problem damals natürlich nicht für die Sprache *Python* gelöst, sondern für die heute nach ihm benannten Turing-Maschinen. Eine Turing-Maschine ist abstrakt gesehen nichts anderes als eine Beschreibung eines Algorithmus. Turing hat also gezeigt, dass es keinen Algorithmus gibt, der entscheiden kann, ob ein gegebener anderer Algorithmus terminiert.

Bemerkung: An dieser Stelle können wir uns fragen, ob es vielleicht eine andere Programmier-Sprache gibt, in der wir das Halte-Problem dann vielleicht doch lösen könnten. Wenn es in dieses Programmier-Sprache Prozeduren, `if`-Verzweigungen und `while`-Schleifen gibt, und wenn wir dort Programm-Texte als Argumente von Funktionen übergeben können, dann ist leicht zu sehen, dass der obige Beweis der Unlösbarkeit des

Halte-Problems sich durch geeignete syntaktische Modifikationen auch auf die andere Programmier-Sprache übertragen lässt.

5.2 Unlösbarkeit des Äquivalenz-Problems

Es gibt noch eine ganze Reihe anderer Funktionen, die nicht berechenbar sind. In der Regel werden wir den Nachweis, dass eine bestimmt Funktion nicht berechenbar ist, indirekt führen und annehmen, dass die gesuchte Funktion doch berechenbar ist. Unter dieser Annahme konstruieren wir dann eine Funktion, die das Halte-Problem löst, was im Widerspruch zu der Unlösbarkeit des Halte-Problems steht. Dieser Widerspruch zwingt uns zu der Folgerung, dass die gesuchte Funktion nicht berechenbar ist. Wir werden dieses Verfahren an einem Beispiel demonstrieren. Vorweg benötigen wir aber noch eine Definition.

Definition 4 (partiell äquivalent, $\simeq$) *Es seien f_1 und f_2 die Namen zwei Python-Funktionen und $a_1, \cdots, a_k$ seien Argumente, mit denen wir diese Funktionen füttern können. Wir definieren*

$$f_1(a_1, \cdots, a_k) \simeq f_2(a_1, \cdots, a_k)$$

g.d.w. einer der beiden folgen Fälle auftritt:

1. $f_1(a_1, \cdots, a_k)\uparrow \quad \wedge \quad f_2(a_1, \cdots, a_k)\uparrow$,
   beide Funktionen divergieren also für die gegebenen Argumente.
2. $\exists r : \Big(f_1(a_1, \cdots, a_k) \rightsquigarrow r \;\wedge\; f_2(a_1, \cdots, a_k) \rightsquigarrow r\Big)$,
   die beiden Funktionen liefern also für die gegebenen Argumente das gleiche Ergebnis.

In diesem Fall sagen wir, dass die beiden Funktions-Aufrufe $f_1(a_1, \cdots, a_k) \simeq f_2(a_1, \cdots, a_k)$ partiell äquivalent sind. □

Wir kommen jetzt zum Äquivalenz-Problem. Die Funktion `equal` sei wie folgt definiert:

```
def equal(t1, t2, a):
    body
```

Zusätzlich erfüllt `equal` die folgende Spezifikation:

1. $t_1 \notin TF \;\vee\; t_2 \notin TF \quad \Leftrightarrow \quad \texttt{equal}(t_1, t_2, a) \rightsquigarrow 2$.
2. Falls
   (a) $t_1 \in TF$ und $name(t_1) = f_1$,
   (b) $t_2 \in TF$ und $name(t_2) = f_2$ und
   (c) $f_1(a) \simeq f_2(a)$

   gilt, dann muss gelten:
   $$\texttt{equal}(t_1, t_2, a) \rightsquigarrow 1.$$
3. Ansonsten gilt
   $$\texttt{equal}(t_1, t_2, a) \rightsquigarrow 0.$$

Wir sagen, dass eine Funktion, die der eben angegebenen Spezifikation genügt, das Äquivalenz-Problem löst.

Theorem 5 (Rice, 1953) *Das Äquivalenz-Problem ist unlösbar.*

```
1   def stops(t, a):
2       l = """
3           def loop(x):
4               while True:
5                   x = 1
6           """
7       e = equal(l, t, a);
8       if e == 2:
9           return 2
10      else:
11          return 1 - e
```

Abbildung 5.3: Eine Implementierung der Funktion `stops`.

Beweis: Wir führen den Beweis indirekt und nehmen an, dass es doch eine Implementierung der Funktion `equal` gibt, die das Äquivalenz-Problem löst. Wir betrachten die in Abbildung 5.3 angegeben Implementierung der Funktion `stops`.

Zu beachten ist, dass in Zeile 3 die Funktion `equal` mit einem String aufgerufen wird, der eine Test-Funktion ist. Diese Test-Funktion hat die folgende Form:

```
def loop(x):
    while True:
        x = 1
```

Es ist offensichtlich, dass diese Funktion für keine Eingabe terminiert. Ist also das Argument t eine Test-Funktion mit Namen f, so liefert die Funktion `equal` immer dann den Wert 1, wenn $f(a)$ nicht terminiert, andernfalls muss sie den Wert 0 zurück geben. Damit liefert die Funktion `stops` aber für eine Test-Funktion t mit Namen f und ein Argument a genau dann 1, wenn der Aufruf $f(a)$ terminiert und würde folglich das Halte-Problem lösen. Das kann nicht sein, also kann es keine Funktion `equal` geben, die das Äquivalenz-Problem löst. □

Die Unlösbarkeit des Äquivalenz-Problems und vieler weiterer praktisch interessanter Probleme folgen aus dem 1953 von Henry G. Rice [Ric53] bewiesenen Satz von Rice.

5.3 Reflexion

Beantworten Sie die folgenden Fragen.

1. Wie ist das Halteproblem definiert?
2. Versuchen Sie, die Definition der Funktion `turing` aus dem Gedächtnis aufzuschreiben und führen Sie dann den Nachweis, dass das Halteproblem nicht lösbar ist.
3. Wie haben wir das Äquivalenz-Problem definiert?
4. Können Sie den Beweis der Unlösbarkeit des Äquivalenz-Problems wiedergeben?

Kapitel 6

Aussagenlogik

6.1 Überblick

Die Aussagenlogik beschäftigt sich mit der Verknüpfung einfacher Aussagen durch Junktoren. Dabei sind Junktoren Worte wie "und", "oder", "nicht", "wenn $\cdots$, dann", und "genau dann, wenn". Unter einfachen Aussagen verstehen Sätze, die

- einen Tatbestand ausdrücken, der entweder wahr oder falsch ist und
- selber keine Junktoren enthalten.

Beispiele für einfache Aussagen sind

1. "*Die Sonne scheint.*"
2. "*Es regnet.*"
3. "*Am Himmel ist ein Regenbogen.*"

Einfache Aussagen dieser Art bezeichnen wir auch als atomare Aussagen, weil sie sich nicht weiter in Teilaussagen zerlegen lassen. Atomare Aussagen lassen sich mit Hilfe der eben angegebenen Junktoren zu zusammengesetzten Aussagen verknüpfen. Ein Beispiel für eine zusammengesetzte Aussage wäre

Wenn die Sonne scheint und es regnet, dann ist ein Regenbogen am Himmel. (1)

Die Aussage ist aus den drei atomaren Aussagen "*Die Sonne scheint.*", "*Es regnet.*", und "*Am Himmel ist ein Regenbogen.*" mit Hilfe der Junktoren "*und*" und "*wenn $\cdots$, dann*" zusammen gesetzt worden. Die Aussagenlogik untersucht, wie sich der Wahrheitswert zusammengesetzter Aussagen aus dem Wahrheitswert der einzelnen Teilaussagen berechnen lässt. Darauf aufbauend wird dann gefragt, in welcher Art und Weise wir aus gegebenen Aussagen neue Aussagen logisch folgern können.

Um die Struktur komplexerer Aussagen übersichtlich darstellen zu können, führen wir in der Aussagenlogik zunächst sogenannte Aussage-Variablen ein. Diese Variablen sind Namen, die für atomare Aussagen stehen. Zusätzlich führen wir für die Junktoren "*nicht*", "*und*", "*oder*", "*wenn, $\cdots$ dann*", und "*genau dann, wenn*" die folgenden Symbole als Abkürzungen ein:

1. $\neg a$ steht für *nicht* a
2. $a \wedge b$ steht für a *und* b
3. $a \vee b$ steht für a *oder* b
4. $a \to b$ steht für *wenn* a, *dann* b
5. $a \leftrightarrow b$ steht für a *genau dann, wenn* b

Aussagenlogische Formeln werden aus Aussage-Variablen mit Hilfe von Junktoren aufgebaut und können beliebig komplex sein. Die Aussage (1) können wir mit Hilfe der Junktoren kürzer als

$$\texttt{SonneScheint} \wedge \texttt{EsRegnet} \rightarrow \texttt{Regenbogen}$$

schreiben. Bestimmte aussagenlogische Formeln sind offenbar immer wahr, egal was wir für die einzelnen Teilaussagen einsetzen. Beispielsweise ist eine Formel der Art

$$p \vee \neg p$$

unabhängig von dem Wahrheitswert der Aussage p immer wahr. Eine aussagenlogische Formel, die immer wahr ist, bezeichnen wir als eine Tautologie. Andere aussagenlogische Formeln sind nie wahr, beispielsweise ist die Formel

$$p \wedge \neg p$$

immer falsch. Eine Formel heißt erfüllbar, wenn es wenigstens eine Möglichkeit gibt, bei der die Formel wahr wird. Im Rahmen der Vorlesung werden wir verschiedene Verfahren entwickeln, mit denen es möglich ist zu entscheiden, ob eine aussagenlogische Formel eine Tautologie ist oder ob Sie wenigstens erfüllbar ist. Solche Verfahren spielen in der Praxis eine wichtige Rolle.

6.2 Anwendungen der Aussagenlogik

Die Aussagenlogik bildet nicht nur die Grundlage für die Prädikatenlogik, sondern sie hat auch wichtige praktische Anwendungen. Aus der großen Zahl der industriellen Anwendungen möchte ich stellvertretend vier Beispiele nennen:

1. Analyse und Design digitaler Schaltungen.

   Komplexe digitale Schaltungen bestehen heute aus Milliarden von logischen Gattern.[1] Ein Gatter ist dabei, aus logischer Sicht betrachtet, ein Baustein, der einen der logischen Junktoren wie "*und*", "*oder*", "*nicht*", etc. auf elektronischer Ebene repräsentiert.

   Die Komplexität solcher Schaltungen wäre ohne den Einsatz rechnergestützter Verfahren zur Verifikation nicht mehr beherrschbar. Die dabei eingesetzten Verfahren sind Anwendungen der Aussagenlogik.

   Eine ganz konkrete Anwendung ist der Schaltungs-Vergleich. Hier werden zwei digitale Schaltungen als aussagenlogische Formeln dargestellt. Anschließend wird versucht, mit aussagenlogischen Mitteln die Äquivalenz dieser Formeln zu zeigen. Software-Werkzeuge, die für die Verifikation digitaler Schaltungen eingesetzt werden, kosten zum Teil mehr als 100 000 $. Die Firma Magma bietet beispielsweise den Equivalence-Checker Quartz Formal zum Preis von 150 000 $ pro Lizenz an. Eine solche Lizenz ist dann drei Jahre lang gültig.

2. Erstellung von Verschlussplänen für die Weichen und Signale von Bahnhöfen.

   Bei einem größeren Bahnhof gibt es einige hundert Weichen und Signale, die ständig neu eingestellt werden müssen, um sogenannte Fahrstraßen für die Züge zu realisieren. Verschiedene Fahrstraßen dürfen sich aus Sicherheitsgründen nicht kreuzen. Die einzelnen Fahrstraßen werden durch sogenannte Verschlusspläne beschrieben. Die Korrektheit solcher Verschlusspläne kann durch aussagenlogische Formeln ausgedrückt werden.

3. Eine Reihe kombinatorischer Puzzles lassen sich als aussagenlogische Formeln kodieren und können dann mit Hilfe aussagenlogischer Methoden gelöst werden. Als ein Beispiel werden wir in der Vorlesung das 8-Damen-Problem behandeln. Dabei geht es um die Frage, ob 8 Damen so auf einem Schachbrett angeordnet werden können, dass keine der Damen eine andere Dame bedroht.

[1] Die Seite `https://en.wikipedia.org/wiki/Transistor_count` gibt einen Überblick über die Komplexität moderner Prozessoren.

6.3 Formale Definition der aussagenlogischen Formeln

Wir behandeln zunächst die Syntax der Aussagenlogik und besprechen anschließend die Semantik. Die Syntax gibt an, wie Formeln geschrieben werden und wie sich Formeln zu Beweisen verknüpfen lassen. Die Semantik befasst sich mit der Bedeutung der Formeln. Nachdem wir die Semantik der aussagenlogischen Formeln mit Hilfe der Mengenlehre definiert haben, zeigen wir anschließend, wie sich diese Semantik in *Python* implementieren lässt.

6.3.1 Syntax der aussagenlogischen Formeln

In diesem Abschnitt legen wir fest, was aussagenlogische Formeln sind: Dazu werden wir aussagenlogische Formeln als Menge von Strings definieren, wobei die Strings in der Menge bestimmte Eigenschaften haben müssen, damit wir von aussagenlogischen Formeln sprechen können.

Zunächst betrachten wir eine Menge $\mathcal{P}$ von sogenannten Aussage-Variablen als gegeben. Typischerweise besteht $\mathcal{P}$ aus der Menge der kleinen lateinischen Buchstaben, die zusätzlich noch indiziert sein dürfen. Beispielsweise werden wir

$$p,\ q,\ r,\ p_1,\ p_2,\ p_3$$

als Aussage-Variablen verwenden. Aussagenlogische Formeln sind dann Wörter, die aus dem Alphabet

$$\mathcal{A} := \mathcal{P} \cup \{\top, \bot, \neg, \vee, \wedge, \rightarrow, \leftrightarrow, (,)\}$$

gebildet werden. Wir definieren die Menge der aussagenlogischen Formeln $\mathcal{F}$ durch Induktion:

1. $\top \in \mathcal{F}$ und $\bot \in \mathcal{F}$.

   Hier steht $\top$ für die Formel, die immer wahr ist, während $\bot$ für die Formel steht, die immer falsch ist. Die Formel $\top$ trägt den Namen Verum[2], für $\bot$ sagen wir Falsum[3].

2. Ist $p \in \mathcal{P}$, so gilt auch $p \in \mathcal{F}$.

   Jede aussagenlogische Variable ist also auch eine aussagenlogische Formel.

3. Ist $f \in \mathcal{F}$, so gilt auch $\neg f \in \mathcal{F}$.

   Die Formel $\neg f$ bezeichnen wir auch als die Negation von f.

4. Sind $f_1, f_2 \in \mathcal{F}$, so gilt auch

$(f_1 \vee f_2) \in \mathcal{F}$	(*gelesen*:	f_1 oder f_2	auch: Disjunktion von f_1 und f_2),
$(f_1 \wedge f_2) \in \mathcal{F}$	(*gelesen*:	f_1 und f_2	auch: Konjunktion von f_1 und f_2),
$(f_1 \rightarrow f_2) \in \mathcal{F}$	(*gelesen*:	wenn f_1, dann f_2	auch: Implikation von f_1 und f_2),
$(f_1 \leftrightarrow f_2) \in \mathcal{F}$	(*gelesen*:	f_1 genau dann, wenn f_2	auch: Bikonditional von f_1 und f_2).

Die Menge $\mathcal{F}$ der aussagenlogischen Formeln ist nun die kleinste Teilmenge der aus dem Alphabet $\mathcal{A}$ gebildeten Wörter, welche die oben aufgelisteten Abschluss-Eigenschaften hat.

Beispiel: Es sei $\mathcal{P} := \{p, q, r\}$. Dann gilt:

1. $p \in \mathcal{F}$,
2. $(p \wedge q) \in \mathcal{F}$,
3. $((\neg p \rightarrow q) \vee (q \rightarrow \neg p)) \in \mathcal{F}$. □

Um Klammern zu sparen, vereinbaren wir die folgenden Regeln:

[2] Verum ist das lateinische Wort für "wahr".
[3] Falsum ist das lateinische Wort für "falsch".

1. Äußere Klammern werden weggelassen, wir schreiben also beispielsweise

   $$p \wedge q \quad \text{statt} \quad (p \wedge q).$$

2. Der Junktor $\neg$ bindet stärker als alle anderen Junktoren.

3. Die Junktoren $\vee$ und $\wedge$ werden implizit links geklammert, d.h. wir schreiben

   $$p \wedge q \wedge r \quad \text{statt} \quad (p \wedge q) \wedge r.$$

   Operatoren, die implizit nach links geklammert werden, nennen wir links-assoziativ.

   Beachten Sie, dass wir für diese Vorlesung vereinbaren, dass die Junktoren $\wedge$ und $\vee$ dieselbe Bindungsstärke haben. Das ist anders als in den Sprache *Python*, denn dort bindet der Operator "`and`" stärker als der Operator "`or`". In der Sprache `C` bindet der Operator "`&&`" ebenfalls stärker als der Operator "`||`".

4. Der Junktor $\rightarrow$ wird implizit rechts geklammert, d.h. wir schreiben

   $$p \rightarrow q \rightarrow r \quad \text{statt} \quad p \rightarrow (q \rightarrow r).$$

   Operatoren, die implizit nach rechts geklammert werden, nennen wir rechts-assoziativ.

5. Die Junktoren $\vee$ und $\wedge$ binden stärker als $\rightarrow$, wir schreiben also

   $$p \wedge q \rightarrow r \quad \text{statt} \quad (p \wedge q) \rightarrow r$$

6. Der Junktor $\rightarrow$ bindet stärker als $\leftrightarrow$, wir schreiben also

   $$p \rightarrow q \leftrightarrow r \quad \text{statt} \quad (p \rightarrow q) \leftrightarrow r.$$

7. Beachten Sie, dass der Junktor $\leftrightarrow$ weder rechts- noch links-assoziativ ist. Daher ist ein Ausdruck der Form

   $$p \leftrightarrow q \leftrightarrow r$$

   undefiniert und muss geklammert werden. Wenn Sie eine solche Formel in einem Buch sehen, ist dies in der Regel als Abkürzung für die Formel

   $$(p \leftrightarrow q) \wedge (q \leftrightarrow r)$$

   zu verstehen.

Bemerkung: Wir werden im Rest dieser Vorlesung eine Reihe von Beweisen führen, bei denen es darum geht, mathematische Aussagen über Formeln nachzuweisen. Bei diesen Beweisen werden wir natürlich ebenfalls aussagenlogische Junktoren wie "genau dann, wenn" oder "wenn $\cdots$, dann" verwenden. Dabei entsteht dann die Gefahr, dass wir die Junktoren, die wir in unseren Beweisen verwenden, mit den Junktoren, die in den aussagenlogischen Formeln auftreten, verwechseln. Um dieses Problem zu umgehen vereinbaren wir:

1. Innerhalb einer aussagenlogischen Formel wird der Junktor "wenn $\cdots$, dann" als "$\rightarrow$" geschrieben.
2. Bei den Beweisen, die wir über aussagenlogische Formeln führen, schreiben wir für diesen Junktor stattdessen "$\Rightarrow$".

Analog wird der Junktor "genau dann, wenn" innerhalb einer aussagenlogischen Formel als "$\leftrightarrow$" geschrieben, aber wenn wir dieser Junktor als Teil eines Beweises verwenden, schreiben wir stattdessen "$\Leftrightarrow$". $\diamond$

6.3.2 Semantik der aussagenlogischen Formeln

In diesem Abschnitt definieren wir die Semantik aussagenlogischer Formeln, wir legen also die Interpretation oder auch Bedeutung dieser Formeln fest. Dazu ordnen wir den aussagenlogischen Formeln Wahrheitswerte zu. Damit diese möglich ist, definieren wir zunächst die Menge $\mathbb{B}$ der Wahrheitswerte:

$$\mathbb{B} := \{\texttt{True}, \texttt{False}\}.$$

Damit können wir nun den Begriff einer aussagenlogischen Interpretation festlegen.

Definition 6 (Aussagenlogische Interpretation) *Eine aussagenlogische Interpretation ist eine Funktion*

$$\mathcal{I} : \mathcal{P} \to \mathbb{B},$$

die jeder Aussage-Variablen $p \in \mathcal{P}$ einen Wahrheitswert $\mathcal{I}(p) \in \mathbb{B}$ zuordnet. ◇

Eine aussagenlogische Interpretation wird oft auch als Belegung der Aussage-Variablen mit Wahrheits-Werten bezeichnet.

Eine aussagenlogische Interpretation $\mathcal{I}$ interpretiert die Aussage-Variablen. Um nicht nur Variablen sondern auch aussagenlogische Formel interpretieren zu können, benötigen wir eine Interpretation der Junktoren "$\neg$", "$\wedge$", "$\vee$", "$\to$" und "$\leftrightarrow$". Zu diesem Zweck definieren wir auf der Menge $\mathbb{B}$ der Wahrheits-Werte Funktionen $\ominus$, $\oslash$, $\odot$, $\ominus$ und $\ominus$, mit deren Hilfe wir die aussagenlogischen Junktoren interpretieren können:

1. $\ominus : \mathbb{B} \to \mathbb{B}$
2. $\oslash : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
3. $\odot : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
4. $\ominus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
5. $\ominus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

Wir könnten die Funktionen $\ominus$, $\oslash$, $\odot$, $\ominus$ und $\ominus$ am einfachsten durch die folgende Tabelle (Tabelle 6.1) definieren:

p	q	$\ominus(p)$	$\odot(p,q)$	$\oslash(p,q)$	$\ominus(p,q)$	$\ominus(p,q)$
`True`	`True`	`False`	`True`	`True`	`True`	`True`
`True`	`False`	`False`	`True`	`False`	`False`	`False`
`False`	`True`	`True`	`True`	`False`	`True`	`False`
`False`	`False`	`True`	`False`	`False`	`True`	`True`

Tabelle 6.1: Interpretation der Junktoren

Nun können wir den Wert, den eine aussagenlogische Formel f unter einer gegebenen aussagenlogischen Interpretation $\mathcal{I}$ annimmt, durch Induktion nach dem Aufbau der Formel f definieren. Wir werden diesen Wert mit $\widehat{\mathcal{I}}(f)$ bezeichnen. Wir setzen:

1. $\widehat{\mathcal{I}}(\bot) := \texttt{False}$.
2. $\widehat{\mathcal{I}}(\top) := \texttt{True}$.
3. $\widehat{\mathcal{I}}(p) := \mathcal{I}(p)$ für alle $p \in \mathcal{P}$.
4. $\widehat{\mathcal{I}}(\neg f) := \ominus\left(\widehat{\mathcal{I}}(f)\right)$ für alle $f \in \mathcal{F}$.
5. $\widehat{\mathcal{I}}(f \wedge g) := \oslash\left(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\right)$ für alle $f, g \in \mathcal{F}$.
6. $\widehat{\mathcal{I}}(f \vee g) := \odot\left(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\right)$ für alle $f, g \in \mathcal{F}$.
7. $\widehat{\mathcal{I}}(f \to g) := \ominus\left(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\right)$ für alle $f, g \in \mathcal{F}$.
8. $\widehat{\mathcal{I}}(f \leftrightarrow g) := \ominus\left(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\right)$ für alle $f, g \in \mathcal{F}$.

Um die Schreibweise nicht übermäßig kompliziert werden zu lassen, unterscheiden wir in Zukunft nicht mehr zwischen der Funktion $\widehat{\mathcal{I}}$ und der Funktion $\mathcal{I}$, wir werden das Hütchen über dem $\mathcal{I}$ also weglassen.

Beispiel: Wir zeigen, wie sich der Wahrheits-Wert der Formel

$$(p \to q) \to (\neg p \to q) \to q$$

für die aussagenlogische Interpretation $\mathcal{I}$, die durch $\mathcal{I}(p) = \texttt{True}$ und $\mathcal{I}(q) = \texttt{False}$ definiert ist, berechnen lässt:

$$\begin{array}{lcl}
\mathcal{I}\Big((p \to q) \to (\neg p \to q) \to q\Big) & = & \ominus\Big(\mathcal{I}\big((p \to q)\big), \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
& = & \ominus\Big(\ominus\big(\mathcal{I}(p), \mathcal{I}(q)\big), \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
& = & \ominus\Big(\ominus\big(\texttt{True}, \texttt{False}\big), \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
& = & \ominus\Big(\texttt{False}, \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
& = & \texttt{True}
\end{array}$$

Beachten Sie, dass wir bei der Berechnung gerade soviele Teile der Formel ausgewertet haben, wie notwendig waren um den Wert der Formel zu bestimmen. Trotzdem ist die eben durchgeführte Rechnung für die Praxis zu umständlich. Stattdessen wird der Wert einer Formel direkt mit Hilfe der Tabelle 6.1 auf Seite 66 berechnet. Wir zeigen exemplarisch, wie wir den Wahrheits-Wert der Formel

$$(p \to q) \to (\neg p \to q) \to q$$

für beliebige Belegungen $\mathcal{I}$ über diese Tabelle berechnen können. Um nun die Wahrheitswerte dieser Formel unter einer gegebenen Belegung der Aussage-Variablen bestimmen zu können, bauen wir eine Tabelle auf, die für jede in der Formel auftretende Teilformel eine Spalte enthält. Tabelle 6.2 auf Seite 67 zeigt die entstehende Tabelle.

p	q	$\neg p$	$p \to q$	$\neg p \to q$	$(\neg p \to q) \to q$	$(p \to q) \to (\neg p \to q) \to q$
True	True	False	True	True	True	True
True	False	False	False	True	False	True
False	True	True	True	True	True	True
False	False	True	True	False	True	True

Tabelle 6.2: Berechnung der Wahrheitswerte von $(p \to q) \to (\neg p \to q) \to q$

Betrachten wir die letzte Spalte der Tabelle so sehen wir, dass dort immer der Wert `True` auftritt. Also liefert die Auswertung der Formel $(p \to q) \to (\neg p \to q) \to q$ für jede aussagenlogische Belegung $\mathcal{I}$ den Wert `True`. Formeln, die immer wahr sind, haben in der Aussagenlogik eine besondere Bedeutung und werden als Tautologien bezeichnet.

Wir erläutern die Aufstellung dieser Tabelle anhand der zweiten Zeile. In dieser Zeile sind zunächst die aussagenlogischen Variablen p auf `True` und q auf `False` gesetzt. Bezeichnen wir die aussagenlogische Interpretation mit $\mathcal{I}$, so gilt also

$$\mathcal{I}(p) = \texttt{True} \text{ und } \mathcal{I}(q) = \texttt{False}.$$

Damit erhalten wir folgende Rechnung:

1. $\mathcal{I}(\neg p) = \ominus(\mathcal{I}(p)) = \ominus(\texttt{True}) = \texttt{False}$
2. $\mathcal{I}(p \to q) = \ominus\,(\mathcal{I}(p), \mathcal{I}(q)) = \ominus\,(\texttt{True}, \texttt{False}) = \texttt{False}$
3. $\mathcal{I}(\neg p \to q) = \ominus\big(\mathcal{I}(\neg p), \mathcal{I}(q)\big) = \ominus(\texttt{False}, \texttt{False}) = \texttt{True}$
4. $\mathcal{I}\big((\neg p \to q) \to q\big) = \ominus\big(\mathcal{I}(\neg p \to q), \mathcal{I}(q)\big) = \ominus(\texttt{True}, \texttt{False}) = \texttt{False}$

5. $\mathcal{I}\big((p \to q) \to (\neg p \to q) \to q\big) = \ominus\big(\mathcal{I}(p \to q), \mathcal{I}((\neg p \to q) \to q)\big) = \ominus\,(\texttt{False}, \texttt{False}) = \texttt{True}$

Für komplexe Formeln ist die Auswertung von Hand viel zu mühsam und fehleranfällig um praktikabel zu sein. Wir zeigen deshalb später, wie sich dieser Prozess mit Hilfe der Sprache *Python* automatisieren lässt.

6.3.3 Extensionale und intensionale Interpretationen der Aussagenlogik

Die Interpretation des aussagenlogischen Junktoren ist rein extensional: Wenn wir den Wahrheitswert der Formel

$$\mathcal{I}(f \to g)$$

berechnen wollen, so müssen wir die Details der Teilformeln f und g nicht kennen, es reicht, wenn wir die Werte $\mathcal{I}(f)$ und $\mathcal{I}(g)$ kennen. Das ist problematisch, denn in der Umgangssprache hat der Junktor "*wenn* $\cdots$, *dann*" auch eine kausale Bedeutung. Mit der extensionalen Implikation wird der Satz

"*Wenn* $3 \cdot 3 = 8$, *dann schneit es.*"

als wahr interpretiert, denn die Formel $3 \cdot 3 = 8$ ist ja falsch. Dass ist problematisch, weil wir diesen Satz in der Umgangssprache als sinnlos erkennen. Insofern ist die extensionale Interpretation des sprachlichen Junktors "*wenn* $\cdots$, *dann*" nur eine Approximation der umgangssprachlichen Interpretation, die sich für die Mathematik und die Informatik aber als ausreichend erwiesen hat.

Es gibt durchaus auch andere Logiken, in denen die Interpretation des Operators "$\to$" von der hier gegebenen Definition abweicht. Solche Logiken werden als intensionale Logiken bezeichnet. Diese Logiken spielen zwar auch in der Informatik eine Rolle, aber da die Untersuchung intensionaler Logiken wesentlich aufwändiger ist als die Untersuchung der extensionalen Logik, werden wir uns auf die Analyse der extensionalen Logik beschränken.

6.3.4 Implementierung in *Python*

Um die bisher eingeführten Begriffe nicht zu abstrakt werden zu lassen, entwickeln wir in *Python* ein Programm, mit dessen Hilfe sich Formeln auswerten lassen. Jedes Mal, wenn wir ein Programm zur Berechnung irgendwelcher Werte entwickeln wollen, müssen wir uns als erstes fragen, wie wir die Argumente der zu implementierenden Funktion und die Ergebnisse dieser Funktion in der verwendeten Programmier-Sprache darstellen können. In diesem Fall müssen wir uns also überlegen, wie wir eine aussagenlogische Formel in *Python* repräsentieren können, denn Ergebniswerte `True` und `False` stehen ja als Wahrheitswerte unmittelbar zur Verfügung. Zusammengesetzte Daten-Strukturen können in *Python* am einfachsten als geschachtelte Tupel dargestellt werden und das ist auch der Weg, den wir für die aussagenlogischen Formeln beschreiten werden. Wir definieren die Repräsentation von aussagenlogischen Formeln formal dadurch, dass wir eine Funktion

$$\mathit{rep} : \mathcal{F} \to \mathit{Python}$$

definieren, die einer aussagenlogischen Formel f ein geschachteltes Tupel $\mathit{rep}(f)$ zuordnet.

1. $\top$ wird repräsentiert durch den String `'\N{up tack}'`, der das Unicode-Zeichen "$\top$" darstellt:

   $\mathit{rep}(\top) :=$ `'\N{up tack}'`.

2. $\bot$ wird repräsentiert durch den String `'\N{down tack}'`, der das Unicode-Zeichen "$\bot$" darstellt:

   $\mathit{rep}(\bot) :=$ `'\N{down tack}'`.

3. Eine aussagenlogische Variable $p \in \mathcal{P}$ repräsentieren wir durch sich selber:

   $\mathit{rep}(p) := p \quad$ für alle $p \in \mathcal{P}$.

4. Ist f eine aussagenlogische Formel, so repräsentieren wir $\neg f$ als verschachteltes Tupel, bei dem wir das Unicode-Zeichen `'\N{not sign}'` verwenden, denn dieses Zeichen wird als "¬" dargestellt:

   $$rep(\neg f) := \big(\texttt{'\N\{not sign\}'}, rep(f)\big).$$

5. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \wedge f_2$ mit Hilfe des Unicode-Zeichens `'\N{logical and}'`:

   $$rep(f \vee g) := \big(\texttt{'\N\{logical and\}'}, rep(f), rep(g)\big).$$

6. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \vee f_2$ mit Hilfe des Unicode-Zeichens `'\N{logical or}'`:

   $$rep(f \vee g) := \big(\texttt{'\N\{logical or\}'}, rep(f), rep(g)\big).$$

7. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \rightarrow f_2$ mit Hilfe des Unicode-Zeichens `'\N{rightwards arrow}'`:

   $$rep(f \rightarrow g) := \big(\texttt{'\N\{rightwards arrow\}'}, rep(f), rep(g)\big).$$

8. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \leftrightarrow f_2$ mit Hilfe des Unicode-Zeichens `'\N{left right arrow}'`:

   $$rep(f \leftrightarrow g) := \big(\texttt{'\N\{left right arrow\}'}, rep(f), rep(g)\big).$$

Bei der Wahl der Repräsentation, mit der wir eine Formel in SETLX repäsentieren, sind wir weitgehend frei. Wir hätten oben sicher auch eine andere Repräsentation verwenden können. Eine gute Repräsentation sollte einerseits möglichst intuitiv sein, andererseits ist es auch wichtig, dass die Repräsentation für die zu entwickelnden Algorithmen adäquat ist. Im Wesentlichen heißt dies, dass es einerseits einfach sein sollte, auf die Komponenten einer Formel zuzugreifen, andererseits sollte es auch leicht sein, die entsprechende Repräsentation zu erzeugen.

Als nächstes geben wir an, wie wir eine aussagenlogische Interpretation in *Python* darstellen. Eine aussagenlogische Interpretation ist eine Funktion

$$\mathcal{I} : \mathcal{P} \rightarrow \mathbb{B}$$

von der Menge der Aussage-Variablen $\mathcal{P}$ in die Menge der Wahrheitswerte $\mathbb{B}$. Die einfachste Möglichkeit eine solche aussagenlogische Interpretation darzustellen besteht darin, dass wir die Menge alle der aussagenlogischen Variablen angeben, die unter der aussagenlogischen Interpretation $\mathcal{I}$ den Wert `True` annehmen:

$$rep(\mathcal{I}) := \big\{x \in \mathcal{P} \mid \mathcal{I}(x) = \texttt{True}\big\}.$$

Damit können wir jetzt eine einfache Funktion implementieren, die den Wahrheitswert einer aussagenlogischen Formel f unter einer gegebenen aussagenlogischen Interpretation $\mathcal{I}$ berechnet. Die Funktion `evaluate.py` ist in Abbildung 6.1 auf Seite 70 gezeigt. Die Funktion `evaluate` erwartet zwei Argumente:

1. Das erste Argument f ist eine aussagenlogische Formel, die als verschachteltes Tupel dargestellt wird.

2. Das zweite Argument I ist eine aussagenlogische Interpretation, die als Menge von aussagenlogischen Variablen dargestellt wird. Für eine aussagenlogische Variable mit dem Namen p können wir den Wert, der dieser Variablen durch I zugeordnet wird, mittels des Ausdrucks p `in` I berechnen.

Wir diskutieren jetzt die Implementierung der Funktion `evaluate()` Zeile für Zeile:

1. Falls die Formel f den Wert $\top$ hat, so ist das Ergebnis der Auswertung unabhängig von der aussagenlogischen Interpretation I immer `True`.

2. Falls die Formel f den Wert $\bot$ hat, so ist das Ergebnis der Auswertung unabhängig von der aussagenlogischen Interpretation I immer `False`.

```
1  def evaluate(F, I):
2      "Evaluate the propositional formula F using the interpretation I"
3      if isinstance(F, str): # F is a propositional variable
4          return F in I
5      if F[0] == '⊤': return True
6      if F[0] == '⊥': return False
7      if F[0] == '¬': return not evaluate(F[1], I)
8      if F[0] == '∧': return evaluate(F[1], I) and evaluate(F[2], I)
9      if F[0] == '∨': return evaluate(F[1], I) or evaluate(F[2], I)
10     if F[0] == '→': return not evaluate(F[1], I) or evaluate(F[2], I)
11     if F[0] == '↔': return evaluate(F[1], I) == evaluate(F[2], I)
```

Abbildung 6.1: Auswertung einer aussagenlogischen Formel

3. In Zeile 5 betrachten wir den Fall, dass das Argument f eine aussagenlogische Variable repräsentiert. Dies erkennen wir daran, dass f ein String ist. die vordefinierte Funktion `isinstance` führt diese Überprüfung durch.

   In diesem Fall müssen wissen, ob die Variable f ein Element der Menge I ist, denn genau dann f ja als wahr interpretiert.

4. In Zeile 7 betrachten wir den Fall, dass f die Form $\neg g$ hat. In diesem Fall werten wir erst g unter der Belegung I aus und negieren dann das Ergebnis.

5. In Zeile 8 betrachten wir den Fall, dass f die Form $g_1 \wedge g_2$ hat. In diesem Fall werten wir zunächst g_1 und g_2 unter der Belegung I aus und verknüpfen das Ergebnis mit dem Operator "`and`".

6. In Zeile 9 betrachten wir den Fall, dass f die Formel $g_1 \vee g_2$ repräsentiert. In diesem Fall werten wir zunächst g_1 und g_2 unter der Belegung I aus und verknüpfen das Ergebnis mit dem Operator "`or`".

7. In Zeile 10 betrachten wir den Fall, dass f die Form $g_1 \to g_2$ hat. In diesem Fall werten wir zunächst g_1 und g_2 unter der Belegung I aus und nutzen dann aus, dass die Formeln

   $$g_1 \to g_2 \quad \text{und} \quad \neg g_1 \vee g_2$$

   äquivalent sind.

8. In Zeile 11 führen wir die Auswertung einer Formel $f \leftrightarrow g$ darauf zurück, dass dieses Formel genau dann wahr ist, wenn f und g den selben Wahrheitswert haben.

6.3.5 Eine Anwendung

Wir betrachten eine spielerische Anwendung der Aussagenlogik. Inspektor Watson wird zu einem Juweliergeschäft gerufen, in das eingebrochen worden ist. In der unmittelbaren Umgebung werden drei Verdächtige Anton, Bruno und Claus festgenommen. Die Auswertung der Akten ergibt folgendes:

1. Einer der drei Verdächtigen muss die Tat begangen haben:

   $$f_1 := a \vee b \vee c.$$

2. Wenn Anton schuldig ist, so hat er genau einen Komplizen.

   Diese Aussage zerlegen wir zunächst in zwei Teilaussagen:

   (a) Wenn Anton schuldig ist, dann hat er mindestens einen Komplizen:

   $$f_2 := a \to b \vee c$$

(b) Wenn Anton schuldig ist, dann hat er höchstens einen Komplizen:

$$f_3 := a \to \neg(b \wedge c)$$

3. Wenn Bruno unschuldig ist, dann ist auch Claus unschuldig:

$$f_4 := \neg b \to \neg c$$

4. Wenn genau zwei schuldig sind, dann ist Claus einer von ihnen.

   Es ist nicht leicht zu sehen, wie diese Aussage sich aussagenlogisch formulieren lässt. Wir behelfen uns mit einem Trick und überlegen uns, wann die obige Aussage falsch ist. Wir sehen, die Aussage ist dann falsch, wenn Claus nicht schuldig ist und wenn gleichzeitig Anton und Bruno schuldig sind. Damit lautet die Formalisierung der obigen Aussage:

$$f_5 := \neg(\neg c \wedge a \wedge b)$$

5. Wenn Claus unschuldig ist, ist Anton schuldig.

$$f_6 := \neg c \to a$$

Wir haben nun eine Menge $F = \{f_1, f_2, f_3, f_4, f_5, f_6\}$ von Formeln. Wir fragen uns nun, für welche Belegungen $\mathcal{I}$ alle Formeln aus der Menge F wahr werden. Wenn es genau eine Belegungen gibt, für die dies der Fall ist, dann liefert uns die Belegung den oder die Täter. Eine Belegung entspricht dabei 1-zu-1 der Menge der Täter. Da es zu zeitraubend ist, alle Belegungen von Hand auszuprobieren, schreiben wir besser ein Programm, das die notwendigen Berechnungen für uns durchführt. Abbildung 6.2 zeigt das Programm `Usual-Suspects.ipynb`. Wir diskutieren diese Programm nun Zeile für Zeile.

```
1  import propLogParser as plp
2
3  def transform(s):
4      "transform the string s into a nested tuple"
5      return plp.LogicParser(s).parse()
6
7  P = { 'a', 'b', 'c' }
8  # Aaron, Bernard, or Caine is guilty.
9  f1 = 'a ∨ b ∨ c'
10 # If Aaron is guilty, he has exactly one accomplice.
11 f2 = 'a → b ∨ c'
12 f3 = 'a → ¬(b ∧ c)'
13 # If Bernard is innocent, then Caine is innocent, too.
14 f4 = '¬b → ¬c'
15 # If exactly two are guilty, then Caine is one of them.
16 f5 = '¬(¬c ∧ a ∧ b)'
17 # If Caine is innocent, then Aaron is guilty.
18 f6 = '¬c → a'
19 Fs = { f1, f2, f3, f4, f5, f6 };
20 Fs = { transform(f) for f in Fs }
21
22 def allTrue(Fs, I):
23     return all({evaluate(f, I) for f in Fs})
24
25 print({ I for I in power(P) if allTrue(Fs, I) })
```

Abbildung 6.2: Programm zur Aufkärung des Einbruchs

1. Da wir die aussagenlogischen Formeln als Strings eingeben, unsere Funktion `evaluate` aber geschachtelte Tupel verarbeitet, importieren wir zunächst den Parser für aussagenlogische Formeln und definieren außerdem die Funktion `transform`, die eine aussagenlogische Formel, die als String vorliegt, in ein geschachteltes Tupel umwandelt.
2. In Zeile 7 definieren wir die Menge P der aussagenlogischen Variablen. Wir benutzen `a` als Abkürzung dafür, dass Aaron schuldig ist, `b` steht für Bernard und `c` ist wahr, wenn Caine schuldig ist.
3. In den Zeilen 7 – 17 definieren wir die Formeln $f_1, \cdots, f_6$.
4. `Fs` ist die Menge aller Formeln.
5. Die Formeln werden in Zeile 20 in geschachtelte Tupel transformiert.
6. Die Funktion `allTrue`(Fs, I) bekommt als Eingabe eine Menge von aussagenlogischen Formeln Fs und eine aussagenlogische Belegung I, die als Teilmenge von P dargestellt wird. Falls alle Formeln f aus der Menge Fs unter der Belegung I wahr sind, gibt diese Funktion als Ergebnis `True` zurück.
7. In Zeile 25 berechnen wir alle Belegungen, für die alle Formeln wahr werden.

Lassen wir das Programm laufen, so sehen wir, dass es nur eine einzige Belegung gibt, bei der alle Formeln wahr werden. Dies ist die Belegung

```
{'b', 'c'}.
```

Damit ist das Problem eindeutig lösbar und Bernard und Caine sind schuldig.

6.4 Tautologien

Die Tabelle in Abbildung 6.2 zeigt, dass die Formel

$$(p \to q) \to (\neg p \to q) \to q$$

für jede aussagenlogische Interpretation wahr ist, denn in der letzten Spalte dieser Tabelle steht immer der Wert `True`. Formeln mit dieser Eigenschaft bezeichnen wir als Tautologie.

Definition 7 (Tautologie) *Ist f eine aussagenlogische Formel und gilt*

$$\mathcal{I}(f) = \texttt{True} \quad \text{für jede aussagenlogische Interpretation } \mathcal{I},$$

dann ist f eine Tautologie. In diesem Fall schreiben wir

$$\models f.$$ ◇

Ist eine Formel f eine Tautologie, so sagen wir auch, dass f allgemeingültig ist.

Beispiele:

1. $\models p \vee \neg p$
2. $\models p \to p$
3. $\models p \wedge q \to p$
4. $\models p \to p \vee q$
5. $\models (p \to \bot) \leftrightarrow \neg p$
6. $\models p \wedge q \leftrightarrow q \wedge p$

Wir können die Tatsache, dass es sich bei diesen Formeln um Tautologien handelt, durch eine Tabelle nachweisen, die analog zu der auf Seite 67 gezeigten Tabelle 6.2 aufgebaut ist. Dieses Verfahren ist zwar konzeptuell sehr einfach, allerdings zu ineffizient, wenn die Anzahl der aussagenlogischen Variablen groß ist. Ziel dieses Kapitels ist daher die Entwicklung eines effizienteren Verfahren.

Die letzten beiden Beispiele in der obigen Aufzählung geben Anlass zu einer neuen Definition.

Definition 8 (Äquivalent) *Zwei Formeln f und g heißen äquivalent g.d.w.*

$$\models f \leftrightarrow g$$

gilt. ◇

Beispiele: Es gelten die folgenden Äquivalenzen:

$\models \neg\bot \leftrightarrow \top$	$\models \neg\top \leftrightarrow \bot$	
$\models p \vee \neg p \leftrightarrow \top$	$\models p \wedge \neg p \leftrightarrow \bot$	Tertium-non-Datur
$\models p \vee \bot \leftrightarrow p$	$\models p \wedge \top \leftrightarrow p$	Neutrales Element
$\models p \vee \top \leftrightarrow \top$	$\models p \wedge \bot \leftrightarrow \bot$	
$\models p \wedge p \leftrightarrow p$	$\models p \vee p \leftrightarrow p$	Idempotenz
$\models p \wedge q \leftrightarrow q \wedge p$	$\models p \vee q \leftrightarrow q \vee p$	Kommutativität
$\models (p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$	$\models (p \vee q) \vee r \leftrightarrow p \vee (q \vee r)$	Assoziativität
$\models \neg\neg p \leftrightarrow p$		Elimination von $\neg\neg$
$\models p \wedge (p \vee q) \leftrightarrow p$	$\models p \vee (p \wedge q) \leftrightarrow p$	Absorption
$\models p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$	$\models p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$	Distributivität
$\models \neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$	$\models \neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$	DeMorgan'sche Regeln
$\models (p \to q) \leftrightarrow \neg p \vee q$		Elimination von $\to$
$\models (p \leftrightarrow q) \leftrightarrow (\neg p \vee q) \wedge (\neg q \vee p)$		Elimination von $\leftrightarrow$

Wir können diese Äquivalenzen nachweisen, indem wir in einer Tabelle sämtliche Belegungen durchprobieren. Eine solche Tabelle heißt auch Wahrheits-Tafel. Wir demonstrieren dieses Verfahren anhand der ersten DeMorgan'schen Regel. Wir erkennen, dass in Abbildung 6.3 in den letzten beiden Spalten in jeder Zeile die-

p	q	$\neg p$	$\neg q$	$p \wedge q$	$\neg(p \wedge q)$	$\neg p \vee \neg q$
`True`	`True`	`False`	`False`	`True`	`False`	`False`
`True`	`False`	`False`	`True`	`False`	`True`	`True`
`False`	`True`	`True`	`False`	`False`	`True`	`True`
`False`	`False`	`True`	`True`	`False`	`True`	`True`

Tabelle 6.3: Nachweis der ersten DeMorgan'schen Regel

selben Werte stehen. Daher sind die Formeln, die zu diesen Spalten gehören, äquivalent.

6.4.1 Testen der Allgemeingültigkeit in *Python*

Die manuelle Überprüfung der Frage, ob eine gegebene Formel f eine Tautologie ist, läuft auf die Erstellung umfangreicher Wahrheitstafeln heraus. Solche Wahrheitstafeln von Hand zu erstellen ist viel zu zeitaufwendig. Wir wollen daher nun ein *Python*-Programm entwickeln, mit dessen Hilfe wir die obige Frage automatisch beantworten können. Die Grundidee ist, dass wir die zu untersuchende Formel für alle möglichen Belegungen auswerten und überprüfen, ob sich bei der Auswertung jedes Mal der Wert `True` ergibt. Dazu müssen wir

zunächst einen Weg finden, alle möglichen Belegungen einer Formel zu berechnen. Wir haben früher schon gesehen, dass Belegungen $\mathcal{I}$ zu Teilmengen M der Menge der aussagenlogischen Variablen $\mathcal{P}$ korrespondieren, denn für jedes $M \subseteq \mathcal{P}$ können wir eine aussagenlogische Belegung $\mathcal{I}(M)$ wie folgt definieren:

$$\mathcal{I}(M)(p) := \begin{cases} \texttt{True} & \text{falls } p \in M; \\ \texttt{False} & \text{falls } p \notin M. \end{cases}$$

Wir stellen daher eine aussagenlogische Belegung durch die Menge $M_{\mathcal{I}}$ der aussagenlogischen Variablen x dar, für die $\mathcal{I}(x)$ den Wert `True` hat. Bei gegebener aussagenlogischer Belegung $\mathcal{I}$ können wir die Menge $M_{\mathcal{I}}$ wie folgt definieren:

$$M_{\mathcal{I}} := \big\{p \in \mathcal{P} \mid \mathcal{I}(p) = \texttt{True}\big\}.$$

Damit können wir nun eine Funktion implementieren, die für eine gegebene aussagenlogische Formel f testet, ob f eine Tautologie ist. Hierzu müssen wir zunächst die Menge $\mathcal{P}$ der aussagenlogischen Variablen bestimmen, die in f auftreten. Abstrakt definieren wir dazu eine Funktion $\texttt{collectVars}(f)$, welche die Menge aller aussagenlogischen Variablen berechnet, die in einer aussagenlogischen Formel f auftreten. Diese Funktion ist durch die folgenden rekursiven Gleichugen spezifiziert:

1. $\texttt{collectVars}(\top) = \{\}$.
2. $\texttt{collectVars}(\bot) = \{\}$.
3. $\texttt{collectVars}(p) = \{p\}$ für alle aussagenlogischen Variablen p.
4. $\texttt{collectVars}(\neg f) := \texttt{collectVars}(f)$.
5. $\texttt{collectVars}(f \wedge g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.
6. $\texttt{collectVars}(f \vee g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.
7. $\texttt{collectVars}(f \rightarrow g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.
8. $\texttt{collectVars}(f \leftrightarrow g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.

Die in Abbildung 6.3 auf Seite 74 zeigt, dass wir diese Gleichungen unmittelbar in *Python* umsetzen können, wobei wir die letzten vier Fälle zusammen gefasst haben, denn die Berechnung der Variablen verläuft in diesen Fällen analog.

```
1  def collectVars(f):
2      "Collect all propositional variables occurring in the formula f."
3      if isinstance(f, str):
4          return { f }
5      if f[0] in ['⊤', '⊥']:
6          return set()
7      if f[0] == '¬':
8          return collectVars(f[1])
9      return collectVars(f[1]) | collectVars(f[2])
```

Abbildung 6.3: Überprüfung der Allgemeingültigkeit einer aussagenlogischen Formel

Damit sind wir nun in der Lage, eine Funktion $\texttt{tautology}(f)$ zu implementieren, die für eine gegebene aussagenlogische f überprüft, ob f eine Tautologie ist. Diese in Abbildung 6.4 auf Seite 75 gezeigte Funktion arbeitet wie folgt:

1. Zunächst berechnen wir die Menge `P` der aussagenlogischen Variablen, die in f auftreten.

2. Sodann berechnen wir mit Hilfe der in dem Modul `power` definierten Funktion `allSubsets` die Liste `A` aller Teilmengen von `P`. Jede als Menge dargestellte aussagenlogische Belegung `I` ist ein Element dieser Liste.
3. Anschließend prüfen wir für jede mögliche Belegung `I`, ob die Auswertung der Formel f für die Belegung `I` den Wert `True` ergibt und geben gegebenenfalls `True` zurück.
4. Andernfalls geben wir die erste Belegung `I` zurück, für welche die Formel f den Wert `False` hat.

```
1  def tautology(f):
2      "Check, whether the formula f is a tautology."
3      P = collectVars(f)
4      A = power.allSubsets(P)
5      if { evaluate(F, I) for I in A } == { True }:
6          return True
7      else:
8          return [I for I in A if not evaluate(F, I)][0]
```

Abbildung 6.4: Überprüfung der Allgemeingültigkeit einer aussagenlogischen Formel

6.4.2 Nachweis der Allgemeingültigkeit durch Äquivalenz-Umformungen

Wollen wir nachweisen, dass eine Formel eine Tautologie ist, können wir uns prinzipiell immer einer Wahrheits-Tafel bedienen. Aber diese Methode hat einen Haken: Kommen in der Formel n verschiedene Aussage-Variablen vor, so hat die Tabelle 2^n Zeilen. Beispielsweise hat die Tabelle zum Nachweis eines der Distributiv-Gesetze bereits 8 Zeilen, da hier 3 verschiedene Variablen auftreten. Das gleiche Problem tritt auch in der im letzten Abschnitt diskutierten Funktion `tautology` auf, denn dort berechnen wir die Potenz-Menge der Menge aller aussagenlogischen Variablen, die in der dort vorgegebenen aussagenlogischen Formel `F` auftreten. Auch hier gilt: Treten in der Formel `F` insgesamt n verschiedene aussagenlogische Variablen auf, so hat die Potenz-Menge 2^n verschiedene Elemente und daher ist dieses Programm für solche Formeln, in denen viele verschiedenen Variablen auftreten, unbrauchbar.

Eine andere Möglichkeit nachzuweisen, dass eine Formel eine Tautologie ist, ergibt sich dadurch, dass wir die Formel mit Hilfe der im letzten Abschnitt aufgeführten Äquivalenzen vereinfachen. Wenn es gelingt, eine Formel F unter Verwendung dieser Äquivalenzen zu $\top$ zu vereinfachen, dann ist gezeigt, dass F eine Tautologie ist. Wir demonstrieren das Verfahren zunächst an einem Beispiel. Mit Hilfe einer Wahrheits-Tafel hatten wir schon gezeigt, dass die Formel

$$(p \to q) \to (\neg p \to q) \to q$$

eine Tautologie ist. Wir zeigen nun, wie wir diesen Tatbestand auch durch eine Kette von Äquivalenz-Umformungen einsehen können:

$$
\begin{array}{lcl}
 & (p \to q) \to (\neg p \to q) \to q & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & (\neg p \vee q) \to (\neg p \to q) \to q & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & (\neg p \vee q) \to (\neg\neg p \vee q) \to q & \text{(Elimination der Doppelnegation)} \\
\Leftrightarrow & (\neg p \vee q) \to (p \vee q) \to q & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & \neg(\neg p \vee q) \vee ((p \vee q) \to q) & \text{(DeMorgan)} \\
\Leftrightarrow & (\neg\neg p \wedge \neg q) \vee ((p \vee q) \to q) & \text{(Elimination der Doppelnegation)} \\
\Leftrightarrow & (p \wedge \neg q) \vee ((p \vee q) \to q) & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & (p \wedge \neg q) \vee (\neg(p \vee q) \vee q) & \text{(DeMorgan)} \\
\Leftrightarrow & (p \wedge \neg q) \vee ((\neg p \wedge \neg q) \vee q) & \text{(Distributivität)} \\
\Leftrightarrow & (p \wedge \neg q) \vee ((\neg p \vee q) \wedge (\neg q \vee q)) & \text{(Tertium-non-Datur)} \\
\Leftrightarrow & (p \wedge \neg q) \vee ((\neg p \vee q) \wedge \top) & \text{(Neutrales Element)} \\
\Leftrightarrow & (p \wedge \neg q) \vee (\neg p \vee q) & \text{(Distributivität)} \\
\Leftrightarrow & (p \vee (\neg p \vee q)) \wedge (\neg q \vee (\neg p \vee q)) & \text{(Assoziativität)} \\
\Leftrightarrow & ((p \vee \neg p) \vee q) \wedge (\neg q \vee (\neg p \vee q)) & \text{(Tertium-non-Datur)} \\
\Leftrightarrow & (\top \vee q) \wedge (\neg q \vee (\neg p \vee q)) & \text{(Neutrales Element)} \\
\Leftrightarrow & \top \wedge (\neg q \vee (\neg p \vee q)) & \text{(Neutrales Element)} \\
\Leftrightarrow & \neg q \vee (\neg p \vee q) & \text{(Assoziativität)} \\
\Leftrightarrow & (\neg q \vee \neg p) \vee q & \text{(Kommutativität)} \\
\Leftrightarrow & (\neg p \vee \neg q) \vee q & \text{(Assoziativität)} \\
\Leftrightarrow & \neg p \vee (\neg q \vee q) & \text{(Tertium-non-Datur)} \\
\Leftrightarrow & \neg p \vee \top & \\
\Leftrightarrow & \top &
\end{array}
$$

Die Umformungen in dem obigen Beweis sind nach einem bestimmten System durchgeführt worden. Um dieses System präzise formulieren zu können, benötigen wir noch einige Definitionen.

Definition 9 (Literal) *Eine aussagenlogische Formel f heißt Literal g.d.w. einer der folgenden Fälle vorliegt:*

1. $f = \top$ *oder* $f = \bot$.
2. $f = p$, *wobei p eine aussagenlogische Variable ist.*
   In diesem Fall sprechen wir von einem positiven Literal.
3. $f = \neg p$, *wobei p eine aussagenlogische Variable ist.*
   In diesem Fall sprechen wir von einem negativen Literal.

Die Menge aller Literale bezeichnen wir mit $\mathcal{L}$. ◇

Später werden wird noch den Begriff des Komplements eines Literals benötigen. Ist l ein Literal, so wird das Komplement von l mit $\overline{l}$ bezeichnet. Das Komplement wird durch Fall-Unterscheidung definiert:

1. $\overline{\top} = \bot$ und $\overline{\bot} = \top$.
2. $\overline{p} := \neg p$, falls $p \in \mathcal{P}$.
3. $\overline{\neg p} := p$, falls $p \in \mathcal{P}$.

Wir sehen, dass das Komplement $\overline{l}$ eines Literals l äquivalent zur Negation von l ist, wir haben also

$$\models \overline{l} \leftrightarrow \neg l.$$

Definition 10 (Klausel) *Eine aussagenlogische Formel K ist eine Klausel wenn K die Form*

$$K = l_1 \vee \cdots \vee l_r$$

hat, wobei l_i für alle $i = 1, \cdots, r$ ein Literal ist. Eine Klausel ist also eine Disjunktion von Literalen. Die Menge aller Klauseln bezeichnen wir mit $\mathcal{K}$. ◇

Oft werden Klauseln auch einfach als Mengen von Literalen betrachtet. Durch diese Sichtweise abstrahieren wir von der Reihenfolge und der Anzahl des Auftretens der Literale in der Disjunktion. Dies ist möglich aufgrund der Assoziativität, Kommutativität und Idempotenz des Junktors "$\vee$". Für die Klausel $l_1 \vee \cdots \vee l_r$ schreiben wir also in Zukunft auch

$$\{l_1, \cdots, l_r\}.$$

Diese Art, eine Klausel als Menge ihrer Literale darzustellen, bezeichnen wir als Mengen-Schreibweise. Das folgende Beispiel illustriert die Nützlichkeit der Mengen-Schreibweise von Klauseln. Wir betrachten die beiden Klauseln

$$p \vee q \vee \neg r \vee p \quad \text{und} \quad \neg r \vee q \vee \neg r \vee p.$$

Die beiden Klauseln sind zwar äquivalent, aber rein syntaktisch sind die Formeln verschieden. Überführen wir die beiden Klauseln in Mengen-Schreibweise, so erhalten wir

$$\{p, q, \neg r\} \quad \text{und} \quad \{\neg r, q, p\}.$$

In einer Menge kommt jedes Element höchstens einmal vor und die Reihenfolge, in der die Elemente auftreten, spielt auch keine Rolle. Daher sind die beiden obigen Mengen gleich! Durch die Tatsache, dass Mengen von der Reihenfolge und der Anzahl der Elemente abstrahieren, implementiert die Mengen-Schreibweise die Assoziativität, Kommutativität und Idempotenz der Disjunktion. Übertragen wir die aussagenlogische Äquivalenz

$$l_1 \vee \cdots \vee l_r \vee \bot \leftrightarrow l_1 \vee \cdots \vee l_r$$

in Mengen-Schreibweise, so erhalten wir

$$\{l_1, \cdots, l_r, \bot\} \leftrightarrow \{l_1, \cdots, l_r\}.$$

Dies zeigt, dass wir das Element $\bot$ in einer Klausel getrost weglassen können. Betrachten wir die letzten Äquivalenz für den Fall, dass $r = 0$ ist, so haben wir

$$\{\bot\} \leftrightarrow \{\}.$$

Damit sehen wir, dass die leere Menge von Literalen als $\bot$ zu interpretieren ist.

Definition 11 *Eine Klausel K ist trivial, wenn einer der beiden folgenden Fälle vorliegt:*

1. $\top \in K$.
2. *Es existiert $p \in \mathcal{P}$ mit $p \in K$ und $\neg p \in K$.*
   In diesem Fall bezeichnen wir p und $\neg p$ als komplementäre Literale. ◇

Satz 12 *Eine Klausel ist genau dann eine Tautologie, wenn sie trivial ist.*

Beweis: Wir nehmen zunächst an, dass die Klausel K trivial ist. Falls nun $\top \in K$ ist, dann gilt wegen der Gültigkeit der Äquivalenz $f \vee \top \leftrightarrow \top$ offenbar $K \leftrightarrow \top$. Ist p eine Aussage-Variable, so dass sowohl $p \in K$ als auch $\neg p \in K$ gilt, dann folgt aufgrund der Äquivalenz $p \vee \neg p \leftrightarrow \top$ sofort $K \leftrightarrow \top$.

Wir nehmen nun an, dass die Klausel K eine Tautologie ist. Wir führen den Beweis indirekt und nehmen an, dass K nicht trivial ist. Damit gilt $\top \notin K$ und K kann auch keine komplementären Literale enthalten. Damit hat K dann die Form

$$K = \{\neg p_1, \cdots, \neg p_m, q_1, \cdots, q_n\} \quad \text{mit } p_i \neq q_j \text{ für alle } i \in \{1, \cdots, m\} \text{ und } j \in \{1, \cdots, n\}.$$

Dann könnten wir eine Interpretation $\mathcal{I}$ wie folgt definieren:

1. $\mathcal{I}(p_i) = \texttt{True}$ für alle $i = 1, \cdots, m$ und
2. $\mathcal{I}(q_j) = \texttt{False}$ für alle $j = 1, \cdots, n$,

Mit dieser Interpretation würde offenbar $\mathcal{I}(K) = \texttt{False}$ gelten und damit könnte K keine Tautologie sein. Also ist die Annahme, dass K nicht trivial ist, falsch. □

Definition 13 (Konjunktive Normalform) *Eine Formel F ist in konjunktiver Normalform (kurz KNF) genau dann, wenn F eine Konjunktion von Klauseln ist, wenn also gilt*

$$F = K_1 \wedge \cdots \wedge K_n,$$

wobei die K_i für alle $i = 1, \cdots, n$ Klauseln sind. ◇

Aus der Definition der KNF folgt sofort:

Korollar 14 *Ist $F = K_1 \wedge \cdots \wedge K_n$ in konjunktiver Normalform, so gilt*

$$\models F \quad \text{genau dann, wenn} \quad \models K_i \quad \text{für alle } i = 1, \cdots, n.$$ □

Damit können wir für eine Formel $F = K_1 \wedge \cdots \wedge K_n$ in konjunktiver Normalform leicht entscheiden, ob F eine Tautologie ist, denn F ist genau dann eine Tautologie, wenn alle Klauseln K_i trivial sind.

Da für die Konjunktion analog zur Disjunktion das Assoziativ-, Kommutativ- und Idempotenz-Gesetz gilt, ist es zweckmäßig, auch für Formeln in konjunktiver Normalform wie folgt eine Mengen-Schreibweise einzuführen: Ist die Formel

$$F = K_1 \wedge \cdots \wedge K_n$$

in konjunktiver Normalform, so repräsentieren wir diese Formel durch die Menge ihrer Klauseln und schreiben

$$F = \{K_1, \cdots, K_n\}.$$

Wir geben ein Beispiel: Sind p, q und r Aussage-Variablen, so ist die Formel

$$(p \vee q \vee \neg r) \wedge (q \vee \neg r \vee p \vee q) \wedge (\neg r \vee p \vee \neg q)$$

in konjunktiver Normalform. In Mengen-Schreibweise wird daraus

$$\big\{\{p, q, \neg r\},\ \{p, \neg q, \neg r\}\big\}.$$

Wir stellen nun ein Verfahren vor, mit dem sich jede Formel F in KNF transformieren lässt. Nach dem oben Gesagten können wir dann leicht entscheiden, ob F eine Tautologie ist.

1. Eliminiere alle Vorkommen des Junktors "$\leftrightarrow$" mit Hilfe der Äquivalenz
   $$(F \leftrightarrow G) \leftrightarrow (F \rightarrow G) \wedge (G \rightarrow F)$$
2. Eliminiere alle Vorkommen des Junktors "$\rightarrow$" mit Hilfe der Äquivalenz
   $$(F \rightarrow G) \leftrightarrow \neg F \vee G$$
3. Schiebe die Negationszeichen soweit es geht nach innen. Verwende dazu die folgenden Äquivalenzen:
   (a) $\neg\bot \leftrightarrow \top$
   (b) $\neg\top \leftrightarrow \bot$
   (c) $\neg\neg F \leftrightarrow F$
   (d) $\neg(F \wedge G) \leftrightarrow \neg F \vee \neg G$
   (e) $\neg(F \vee G) \leftrightarrow \neg F \wedge \neg G$

   In dem Ergebnis, das wir nach diesem Schritt erhalten, stehen die Negationszeichen nur noch unmittelbar vor den aussagenlogischen Variablen. Formeln mit dieser Eigenschaft bezeichnen wir auch als Formeln in Negations-Normalform.
4. Stehen in der Formel jetzt "$\vee$"-Junktoren über "$\wedge$"-Junktoren, so können wir durch Ausmultiplizieren, sprich Verwendung der Äquivalenz
   $$\begin{aligned} & (F_1 \wedge \cdots \wedge F_m) \vee (G_1 \wedge \cdots \wedge G_n) \\ \leftrightarrow\ & (F_1 \vee G_1) \wedge \cdots \wedge (F_1 \vee G_n)\ \wedge\ \cdots\ \wedge\ (F_m \vee G_1) \wedge \cdots \wedge (F_m \vee G_n) \end{aligned}$$

den Junktor "$\vee$" nach innen schieben.

5. In einem letzten Schritt überführen wir die Formel nun in Mengen-Schreibweise, indem wir zunächst die Disjunktionen aller Literale als Mengen zusammenfassen und anschließend alle so entstandenen Klauseln wieder in einer Menge zusammen fassen.

Hier sollten wir noch bemerken, dass die Formel beim Ausmultiplizieren stark anwachsen kann. Das liegt daran, dass die Formel F auf der rechten Seite der Äquivalenz $F \vee (G \wedge H) \leftrightarrow (F \vee G) \wedge (F \vee H)$ zweimal auftritt, während sie links nur einmal vorkommt.

Wir demonstrieren das Verfahren am Beispiel der Formel

$$(p \to q) \to (\neg p \to \neg q).$$

1. Da die Formel den Junktor "$\leftrightarrow$" nicht enthält, ist im ersten Schritt nichts zu tun.

2. Die Elimination von "$\to$" liefert

   $$\neg(\neg p \vee q) \vee (\neg\neg p \vee \neg q).$$

3. Die Umrechnung auf Negations-Normalform liefert

   $$(p \wedge \neg q) \vee (p \vee \neg q).$$

4. Durch "Ausmultiplizieren" erhalten wir

   $$\big(p \vee (p \vee \neg q)\big) \wedge \big(\neg q \vee (p \vee \neg q)\big).$$

5. Die Überführung in die Mengen-Schreibweise ergibt zunächst als Klauseln die beiden Mengen

   $$\{p, p, \neg q\} \quad \text{und} \quad \{\neg q, p, \neg q\}.$$

   Da die Reihenfolge der Elemente einer Menge aber unwichtig ist und außerdem eine Menge jedes Element nur einmal enthält, stellen wir fest, dass diese beiden Klauseln gleich sind. Fassen wir jetzt die Klauseln noch in einer Menge zusammen, so erhalten wir

   $$\big\{\{p, \neg q\}\big\}.$$

   Beachten Sie, dass sich die Formel durch die Überführung in Mengen-Schreibweise noch einmal deutlich vereinfacht hat.

Damit ist die Formel in KNF überführt.

6.4.3 Berechnung der konjunktiven Normalform in *Python*

Wir geben nun eine Reihe von Funktionen an, mit deren Hilfe sich eine gegebene Formel f in konjunktive Normalform überführen lässt. Diese Funktionen sind Teil des Jupyter Notebooks CNF.ipynb. Wir beginnen mit der Funktion

$$\texttt{elimBiconditional} : \mathcal{F} \to \mathcal{F}$$

welche die Aufgabe hat, eine vorgegebene aussagenlogische Formel f in eine äquivalente Formel umzuformen, die den Junktor "$\leftrightarrow$" nicht mehr enthält. Die Funktion $\texttt{elimBiconditional}(f)$ wird durch Induktion über den Aufbau der aussagenlogischen Formel f definiert. Dazu stellen wir zunächst rekursive Gleichungen auf, die das Verhalten der Funktion `elimBiconditional` beschreiben:

1. Wenn f eine Aussage-Variable p ist, so ist nichts zu tun:

   $$\texttt{elimBiconditional}(p) = p \quad \text{für alle } p \in \mathcal{P}.$$

2. Die Fälle, in denen f gleich dem Verum oder dem Falsum ist, sind ebenfalls trivial:

   $$\texttt{elimBiconditional}(\top) = \top \quad \text{und} \quad \texttt{elimBiconditional}(\bot) = \bot.$$

3. Hat f die Form $f = \neg g$, so eliminieren wir den Junktor "$\leftrightarrow$" rekursiv aus der Formel g und negieren die resultierende Formel:

   $$\texttt{elimBiconditional}(\neg g) = \neg\texttt{elimBiconditional}(g).$$

4. Im Falle $f = g_1 \wedge g_2$ eliminieren wir rekursiv den Junktor "$\leftrightarrow$" aus den Formeln g_1 und g_2:

   $$\texttt{elimBiconditional}(g_1 \wedge g_2) = \texttt{elimBiconditional}(g_1) \wedge \texttt{elimBiconditional}(g_2).$$

5. Im Falle $f = g_1 \vee g_2$ eliminieren wir den Junktor "$\leftrightarrow$" aus den Formeln g_1 und g_2:

   $$\texttt{elimBiconditional}(g_1 \vee g_2) = \texttt{elimBiconditional}(g_1) \vee \texttt{elimBiconditional}(g_2).$$

6. Im Falle $f = g_1 \rightarrow g_2$ eliminieren wir den Junktor "$\leftrightarrow$" aus den Formeln g_1 und g_2:

   $$\texttt{elimBiconditional}(g_1 \rightarrow g_2) = \texttt{elimBiconditional}(g_1) \rightarrow \texttt{elimBiconditional}(g_2).$$

7. Hat f die Form $f = g_1 \leftrightarrow g_2$, so benutzen wir die Äquivalenz

   $$(g_1 \leftrightarrow g_2) \leftrightarrow \big((g_1 \rightarrow g_2) \wedge (g_2 \rightarrow g_1)\big).$$

   Das führt auf die Gleichung:

   $$\texttt{elimBiconditional}(g_1 \leftrightarrow g_2) = \texttt{elimBiconditional}\big((g_1 \rightarrow g_2) \wedge (g_2 \rightarrow g_1)\big).$$

   Der Aufruf der Funktion `elimBiconditional` auf der rechten Seite der Gleichung ist notwendig, denn der Junktor "$\leftrightarrow$" kann ja noch in g_1 und g_2 auftreten.

```
1  def elimBiconditional(f):
2      "Eliminate the logical operator '↔' from the formula f."
3      if isinstance(f, str):   # This case covers variables.
4          return f
5      if f[0] == '↔':
6          g, h = f[1:]
7          ge   = elimBiconditional(g)
8          he   = elimBiconditional(h)
9          return ('∧', ('→', ge, he), ('→', he, ge))
10     if f[0] == '⊤':
11         return f
12     if f[0] == '⊥':
13         return f
14     if f[0] == '¬':
15         g  = f[1]
16         ge = elimBiconditional(g)
17         return ('¬', ge)
18     else:
19         op, g, h = f
20         ge       = elimBiconditional(g)
21         he       = elimBiconditional(h)
22         return (op, ge, he)
```

Abbildung 6.5: Elimination von $\leftrightarrow$

Abbildung 6.5 auf Seite 80 zeigt die Implementierung der Funktion `elimBiconditional`.

1. In Zeile 3 prüft der Funktions-Aufruf $\texttt{isinstance}(f, \texttt{str})$, ob f ein String ist. In diesem Fall muss f eine aussagenlogische Variable sein, denn alle anderen aussagenlogischen Formeln werden als geschachtelte Listen dargestellt. Daher wird f in diesem Fall unverändert zurück gegeben.

2. In Zeile 10 und 12 behandeln wir die Fälle, dass f gleich dem Verum oder dem Falsum ist. Hier ist zu beachten, dass diese Formeln ebenfalls als geschachtelte Tupel dargestellt werden, Verum wird beispielsweise als das Tuple `('⊤',)` dargestellt, während Falsum von uns in *Python* durch das Tuple `('⊥',)` repräsentiert wird. Auch in diesem Fall wird f unverändert zurück gegeben.

3. In Zeile 14 betrachten wir den Fall, dass f eine Negation ist. Dann hat f Form

   `('¬', g)`

   und wir müssen den Junktor "$\leftrightarrow$" rekursiv aus g entfernen.

4. In den jetzt noch verbleibenden Fällen hat f die Form

   `('o', g, h)` mit $o \in \{\rightarrow, \wedge, \vee\}$.

   In diesen Fällen muss der Junktor "$\leftrightarrow$" rekursiv aus den Teilformeln g und h entfernt werden.

Als nächstes betrachten wir die Funktion zur Elimination des Junktors "$\rightarrow$". Abbildung 6.6 auf Seite 81 zeigt die Implementierung der Funktion `elimFolgt`. Die der Implementierung zu Grunde liegende Idee ist dieselbe wie bei der Elimination des Junktors "$\leftrightarrow$". Der einzige Unterschied besteht darin, dass wir jetzt die Äquivalenz

$$(G_1 \rightarrow G_2) \leftrightarrow (\neg G_1 \vee G_2)$$

benutzen. Außerdem können wir bei der Implementierung dieser Funktion voraussetzen, dass der Junktor "$\leftrightarrow$" bereits aus der aussagenlogischen Formel `F`, die als Argument übergeben wird, eliminiert worden ist. Dadurch entfällt bei der Implementierung ein Fall.

```
1  def elimConditional(f):
2      "Eliminate the logical operator '→' from f."
3      if isinstance(f, str):
4          return f
5      if f[0] == '⊤':
6          return f
7      if f[0] == '⊥':
8          return f
9      if f[0] == '→':
10         g, h = f[1:]
11         ge   = elimConditional(g)
12         he   = elimConditional(h)
13         return ('∨', ('¬', ge), he)
14     if f[0] == '¬':
15         g  = f[1]
16         ge = elimConditional(g)
17         return ('¬', ge)
18     else:
19         op, g, h = f
20         ge       = elimConditional(g)
21         he       = elimConditional(h)
22         return (op, ge, he)
```

Abbildung 6.6: Elimination von $\rightarrow$

Als nächstes zeigen wir die Funktionen zur Berechnung der Negations-Normalform. Abbildung 6.7 auf Seite 83 zeigt die Implementierung der Funktionen `nnf` und `neg`, die sich wechselseitig aufrufen. Dabei berechnet `nnf(f)` die Negations-Normalform von f, während `neg(f)` die Negations-Normalform von $\neg f$ berechnet, es gilt also

$$\texttt{neg}(\texttt{F}) = \texttt{nnf}(\neg \texttt{f}).$$

Die eigentliche Arbeit wird dabei in der Funktion neg erledigt, denn dort werden die beiden DeMorgan'schen Gesetze

$$\neg(f \wedge g) \leftrightarrow (\neg f \vee \neg g) \quad \text{und} \quad \neg(f \vee g) \leftrightarrow (\neg f \wedge \neg g)$$

angewendet. Wir beschreiben die Umformung in Negations-Normalform durch die folgenden Gleichungen:

1. $\texttt{nnf}(p) = p$ für alle $p \in \mathcal{P}$,
2. $\texttt{nnf}(\top) = \top$,
3. $\texttt{nnf}(\bot) = \bot$,
4. $\texttt{nnf}(\neg \texttt{f}) = \texttt{neg}(\texttt{f})$,
5. $\texttt{nnf}(f_1 \wedge f_2) = \texttt{nnf}(f_1) \wedge \texttt{nnf}(f_2)$,
6. $\texttt{nnf}(f_1 \vee f_2) = \texttt{nnf}(f_1) \vee \texttt{nnf}(f_2)$.

Die Hilfsprozedur neg, die die Negations-Normalform von $\neg f$ berechnet, spezifizieren wir ebenfalls durch rekursive Gleichungen:

1. $\texttt{neg}(p) = \texttt{nnf}(\neg p) = \neg p$ für alle Aussage-Variablen p.
2. $\texttt{neg}(\top) = \texttt{nnf}(\neg\top) = \texttt{nnf}(\bot) = \bot$,
3. $\texttt{neg}(\bot) = \texttt{nnf}(\neg\bot) = \texttt{nnf}(\top) = \top$,
4. $\texttt{neg}(\neg f) = \texttt{nnf}(\neg\neg f) = \texttt{nnf}(f)$.
5. $$\begin{array}{ll} & \texttt{neg}(f_1 \wedge f_2) \\ = & \texttt{nnf}(\neg(f_1 \wedge f_2)) \\ = & \texttt{nnf}(\neg f_1 \vee \neg f_2) \\ = & \texttt{nnf}(\neg f_1) \vee \texttt{nnf}(\neg f_2) \\ = & \texttt{neg}(f_1) \vee \texttt{neg}(f_2). \end{array}$$
   Also haben wir:
   $$\texttt{neg}(f_1 \wedge f_2) = \texttt{neg}(f_1) \vee \texttt{neg}(f_2).$$
6. $$\begin{array}{ll} & \texttt{neg}(f_1 \vee f_2) \\ = & \texttt{nnf}(\neg(f_1 \vee f_2)) \\ = & \texttt{nnf}(\neg f_1 \wedge \neg f_2) \\ = & \texttt{nnf}(\neg f_1) \wedge \texttt{nnf}(\neg f_2) \\ = & \texttt{neg}(f_1) \wedge \texttt{neg}(f_2). \end{array}$$
   Also haben wir:
   $$\texttt{neg}(f_1 \vee f_2) = \texttt{neg}(f_1) \wedge \texttt{neg}(f_2).$$

Die in Abbildung 6.7 auf Seite 83 gezeigten Funktionen setzen die oben diskutierten Gleichungen unmittelbar um.

```
def nnf(f):
    "Compute the negation normal form of f."
    if isinstance(f, str):
        return f
    if f[0] == '⊤':
        return f
    if f[0] == '⊥':
        return f
    if f[0] == '¬':
        g = f[1]
        return neg(g)
    if f[0] == '∧':
        g, h = f[1:]
        return ('∧', nnf(g), nnf(h))
    if f[0] == '∨':
        g, h = f[1:]
        return ('∨', nnf(g), nnf(h))

def neg(f):
    "Compute the negation normal form of ¬f."
    if isinstance(f, str):
        return ('¬', f)
    if f[0] == '⊤':
        return ('⊥',)
    if f[0] == '⊥':
        return ('⊤')
    if f[0] == '¬':
        g = f[1]
        return nnf(g)
    if f[0] == '∧':
        g, h = f[1:]
        return ('∨', nnf(g), nnf(h))
    if f[0] == '∨':
        g, h = f[1:]
        return ('∧', nnf(g), nnf(h))
```

Abbildung 6.7: Berechnung der Negations-Normalform

Als letztes stellen wir die Funktionen vor, mit denen die Formeln, die bereits in Negations-Normalform sind, ausmultipliziert und dadurch in konjunktive Normalform gebracht werden. Gleichzeitig werden die zu normalisierenden Formeln dabei in die Mengen-Schreibweise transformiert, d.h. die Formeln werden als Mengen von Mengen von Literalen dargestellt. Dabei interpretieren wir eine Menge von Literalen als Disjunktion der Literale und eine Menge von Klauseln interpretieren wir als Konjunktion der Klauseln. Mathematisch ist unser Ziel also, eine Funktion

$$\texttt{cnf} : \texttt{NNF} \rightarrow \texttt{KNF}$$

zu definieren, so dass $\texttt{cnf}(f)$ für eine Formel f, die in Negations-Normalform vorliegt, eine Menge von Klauseln als Ergebnis zurück gibt, deren Konjunktion zu f äquivalent ist. Die Definition von $\texttt{cnf}(f)$ erfolgt rekursiv.

1. Falls f eine aussagenlogische Variable ist, geben wir als Ergebnis eine Menge zurück, die genau eine Klausel enthält. Diese Klausel ist selbst wieder eine Menge von Literalen, die als einziges Literal die aussagenlogische Variable f enthält:

$$\texttt{cnf}(f) := \{\{f\}\} \quad \text{falls } f \in \mathcal{P}.$$

2. Wir hatten früher gesehen, dass die leere Menge von *Klauseln* als $\top$ interpretiert werden kann. Daher gilt:

   $$\texttt{cnf}(\top) := \{\}.$$

3. Wir hatten früher gesehen, dass die leere Menge von *Literalen* als $\bot$ interpretiert werden kann. Daher gilt:

   $$\texttt{cnf}(\bot) := \{\{\}\}.$$

4. Falls f eine Negation ist, dann muss gelten

   $$f = \neg p \quad \text{mit } p \in \mathcal{P},$$

   denn f ist ja in Negations-Normalform und in einer solchen Formel kann der Negations-Operator nur auf eine aussagenlogische Variable angewendet werden. Daher ist f ein Literal und wir geben als Ergebnis eine Menge zurück, die genau eine Klausel enthält. Diese Klausel ist selbst wieder eine Menge von Literalen, die als einziges Literal die Formel f enthält:

   $$\texttt{cnf}(\neg p) := \{\{\neg p\}\} \quad \text{falls } p \in \mathcal{P}.$$

5. Falls f eine Konjunktion ist und also $f = g \wedge h$ gilt, dann können wir die zunächst die Formeln g und h in KNF transformieren. Dabei erhalten wir dann Mengen von Klauseln $\texttt{cnf}(g)$ und $\texttt{cnf}(h)$. Da wir eine Menge von Klauseln als Konjunktion der in der Menge enthaltenen Klauseln interpretieren, reicht es aus, die Vereinigung der Mengen $\texttt{cnf}(f)$ und $\texttt{cnf}(g)$ zu bilden, wir haben also

   $$\texttt{cnf}(g \wedge h) = \texttt{cnf}(g) \cup \texttt{cnf}(h).$$

6. Falls $f = g \vee h$ ist, transformieren wir zunächst g und h in KNF. Dabei erhalten wir

   $$\texttt{cnf}(g) = \{g_1, \cdots, g_m\} \quad \text{und} \quad \texttt{cnf}(h) = \{h_1, \cdots, h_n\}.$$

   Dabei sind die g_i und die h_j Klauseln. Um nun die KNF von $g \vee h$ zu bilden, rechnen wir wie folgt:

   $$\begin{array}{ll} & g \vee h \\ \Leftrightarrow & (k_1 \wedge \cdots \wedge k_m) \vee (l_1 \wedge \cdots \wedge l_n) \\ \Leftrightarrow & (k_1 \vee l_1) \quad \wedge \quad \cdots \quad \wedge \quad (k_m \vee l_1) \quad \wedge \\ & \quad \vdots \qquad\qquad\qquad\qquad\quad \vdots \\ & (k_1 \vee l_n) \quad \wedge \quad \cdots \quad \wedge \quad (k_m \vee l_n) \\ \Leftrightarrow & \{k_i \vee l_j : i \in \{1, \cdots, m\}, j \in \{1, \cdots, n\}\} \end{array}$$

   Berücksichtigen wir noch, dass Klauseln in der Mengen-Schreibweise als Mengen von Literalen aufgefasst werden, die implizit disjunktiv verknüpft werden, so können wir für $k_i \vee l_j$ auch $k_i \cup l_j$ schreiben. Insgesamt erhalten wir damit

   $$\texttt{cnf}(g \vee h) = \big\{k \cup l \mid k \in \texttt{cnf}(g) \ \wedge \ l \in \texttt{cnf}(h)\big\}.$$

Abbildung 6.8 auf Seite 85 zeigt die Implementierung der Funktion `cnf`. (Der Name `cnf` ist die Abkürzung von *conjunctive normal form*.)

Zum Abschluss zeigen wir in Abbildung 6.9 auf Seite 85 wie die einzelnen Funktionen zusammenspielen.

1. Die Funktion `normalize` eliminiert zunächst die Junktoren "$\leftrightarrow$" mit Hilfe der Funktion `elimBiconditional`.
2. Anschließend wird der Junktor "$\rightarrow$" mit Hilfe der Funktion `elimConditional` ersetzt.
3. Der Aufruf von `nnf` bringt die Formel in Negations-Normalform.
4. Die Negations-Normalform wird nun mit Hilfe der Funktion `cnf` in konjunktive Normalform gebracht, wobei gleichzeitig die Formel in Mengen-Schreibweise überführt wird.

```
def cnf(f):
    if isinstance(f, str):
        return { frozenset({f}) }
    if f[0] == '⊤':
        return set()
    if f[0] == '⊥':
        return { frozenset() }
    if f[0] == '¬':
        return { frozenset({f}) }
    if f[0] == '∧':
        g, h = f[1:]
        return cnf(g) | cnf(h)
    if f[0] == '∨':
        g, h = f[1:]
        return { k1 | k2 for k1 in cnf(g) for k2 in cnf(h) }
```

Abbildung 6.8: Berechnung der konjunktiven Normalform

5. Schließlich entfernt die Funktion `simplify` alle Klauseln aus der Menge `N4`, die trivial sind.
6. Die Funktion `isTrivial` überprüft, ob eine Klausel C, die in Mengen-Schreibweise vorliegt, sowohl eine Variable p als auch die Negation $\neg p$ dieser Variablen enthält, denn dann ist diese Klausel zu $\top$ äquivalent und kann weggelassen werden.

Das vollständige Programm zur Berechnung der konjunktiven Normalform finden Sie als die Datei `knf.stlx` unter GitHub.

```
def normalize (f):
    n1 = elimBiconditional(f)
    n2 = elimConditional(n1)
    n3 = nnf(n2)
    n4 = cnf(n3)
    return simplify(n4)

def simplify(Clauses):
    return { C for C in Clauses if not isTrivial(C) }

def isTrivial(Clause):
    return any(('¬', p) in Clause for p in Clause)
```

Abbildung 6.9: Normalisierung einer Formel

Aufgabe 4: Berechnen Sie die konjunktiven Normalformen der folgenden aussagenlogischen Formeln und geben Sie Ihr Ergebnis in Mengenschreibweise an. Überprüfen Sie Ihr Ergebnis mit Hilfe des Jupyter-Notebooks CNF.ipynb.

1. $p \vee q \rightarrow r$,
2. $p \vee q \leftrightarrow r$,
3. $(p \rightarrow q) \leftrightarrow (\neg p \rightarrow \neg q)$,
4. $(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$,

5. $\neg r \wedge (q \vee p \to r) \to \neg q \wedge \neg p$.

6.5 Der Herleitungs-Begriff

Ist $\{f_1, \cdots, f_n\}$ eine Menge von Formeln, und g eine weitere Formel, so können wir uns fragen, ob die Formel g aus $f_1, \cdots, f_n$ folgt, ob also

$$\models f_1 \wedge \cdots \wedge f_n \to g$$

gilt. Es gibt verschiedene Möglichkeiten, diese Frage zu beantworten. Ein Verfahren kennen wir schon: Zunächst überführen wir die Formel $f_1 \wedge \cdots \wedge f_n \to g$ in konjunktive Normalform. Wir erhalten dann eine Menge $\{k_1, \cdots, k_m\}$ von Klauseln, deren Konjunktion zu der Formel

$$f_1 \wedge \cdots \wedge f_n \to g$$

äquivalent ist. Diese Formel ist nun genau dann eine Tautologie, wenn jede der Klauseln $k_1, \cdots, k_m$ trivial ist.

Das oben dargestellte Verfahren ist aber sehr aufwendig. Wir zeigen dies anhand eines Beispiels und wenden das Verfahren an, um zu entscheiden, ob $p \to r$ aus den beiden Formeln $p \to q$ und $q \to r$ folgt. Wir bilden also die konjunktive Normalform der Formel

$$h := (p \to q) \wedge (q \to r) \to p \to r$$

und erhalten nach mühsamer Rechnung

$$(p \vee \neg p \vee r \vee \neg r) \wedge (\neg q \vee \neg p \vee r \vee \neg r) \wedge (\neg q \vee \neg p \vee q \vee r) \wedge (p \vee \neg p \vee q \vee r).$$

Zwar können wir jetzt sehen, dass die Formel h eine Tautologie ist, aber angesichts der Tatsache, dass wir mit bloßem Auge sehen, dass $p \to r$ aus den Formeln $p \to q$ und $q \to r$ folgt, ist die Rechnung doch sehr aufwendig.

Wir stellen daher nun eine weiteres Verfahren vor, mit dessen Hilfe wir entscheiden können, ob eine Formel aus einer gegebenen Menge von Formeln folgt. Die Idee bei diesem Verfahren ist es, die Formel f mit Hilfe von Schluss-Regeln aus den gegebenen Formeln $f_1, \cdots, f_n$ herzuleiten. Das Konzept einer Schluss-Regel wird in der nun folgenden Definition festgelegt.

Definition 15 (Schluss-Regel) *Eine Schluss-Regel ist eine Paar $\langle\langle f_1, \cdots, f_n\rangle, k\rangle$. dabei ist $\langle f_1, \cdots, f_n\rangle$ ein Tupel von Formeln und k ist eine einzelne Formel. Die Formeln $f_1, \cdots, f_n$ bezeichnen wir als Prämissen, die Formel k heißt die Konklusion der Schluss-Regel. Ist das Paar $\langle\langle f_1, \cdots, f_n\rangle, k\rangle$ eine Schluss-Regel, so schreiben wir dies als:*

$$\frac{f_1 \quad \cdots \quad f_n}{k}.$$

Wir lesen diese Schluss-Regel wie folgt: "Aus $f_1, \cdots, f_n$ kann auf k geschlossen werden." ◇

Beispiele für Schluss-Regeln:

Modus Ponens	Modus Tollens	Unfug
$\dfrac{f \quad f \to g}{g}$	$\dfrac{\neg g \quad f \to g}{\neg f}$	$\dfrac{\neg f \quad f \to g}{\neg g}$

Die Definition der Schluss-Regel schränkt zunächst die Formeln, die als Prämissen bzw. Konklusion verwendet werden können, nicht weiter ein. Es ist aber sicher nicht sinnvoll, beliebige Schluss-Regeln zuzulassen. Wollen wir Schluss-Regeln in Beweisen verwenden, so sollten die Schluss-Regeln in dem in der folgenden Definition erklärten Sinne korrekt sein.

Definition 16 (Korrekte Schluss-Regel) *Eine Schluss-Regel der Form*

$$\frac{f_1 \quad \cdots \quad f_n}{k}$$

ist genau dann korrekt, wenn $\models f_1 \wedge \cdots \wedge f_n \rightarrow k$ *gilt.* ◇

Mit dieser Definition sehen wir, dass die oben als "Modus Ponens" und "Modus Tollens" bezeichneten Schluss-Regeln korrekt sind, während die als "Unfug" bezeichnete Schluss-Regel nicht korrekt ist.

Im Folgenden gehen wir davon aus, dass alle Formeln Klauseln sind. Einerseits ist dies keine echte Einschränkung, denn wir können ja jede Formel in eine äquivalente Menge von Klauseln umformen. Andererseits haben die Formeln bei vielen in der Praxis auftretenden aussagenlogischen Problemen ohnehin die Gestalt von Klauseln. Daher stellen wir jetzt eine Schluss-Regel vor, in der sowohl die Prämissen als auch die Konklusion Klauseln sind.

Definition 17 (Schnitt-Regel) *Ist p eine aussagenlogische Variable und sind k_1 und k_2 Mengen von Literalen, die wir als Klauseln interpretieren, so bezeichnen wir die folgende Schluss-Regel als die Schnitt-Regel:*

$$\frac{k_1 \cup \{p\} \quad \{\neg p\} \cup k_2}{k_1 \cup k_2}.$$ ◇

Die Schnitt-Regel ist sehr allgemein. Setzen wir in der obigen Definition für $k_1 = \{\}$ und $k_2 = \{q\}$ ein, so erhalten wir die folgende Regel als Spezialfall:

$$\frac{\{\} \cup \{p\} \qquad \{\neg p\} \cup \{q\}}{\{\} \cup \{q\}}$$

Interpretieren wir nun die Mengen als Disjunktionen, so haben wir:

$$\frac{p \qquad \neg p \vee q}{q}$$

Wenn wir jetzt noch berücksichtigen, dass die Formel $\neg p \vee q$ äquivalent zu der Formel $p \rightarrow q$ ist, dann ist das nichts anderes als Modus Ponens. Die Regel Modus Tollens ist ebenfalls ein Spezialfall der Schnitt-Regel. Wir erhalten diese Regel, wenn wir in der Schnitt-Regel $k_1 = \{\neg q\}$ und $k_2 = \{\}$ setzen.

Satz 18 *Die Schnitt-Regel ist korrekt.*

Beweis: Wir müssen zeigen, dass

$$\models (k_1 \vee p) \wedge (\neg p \vee k_2) \rightarrow k_1 \vee k_2$$

gilt. Dazu überführen wir die obige Formel in konjunktive Normalform:

$$\begin{array}{ll}
 & (k_1 \vee p) \wedge (\neg p \vee k_2) \rightarrow k_1 \vee k_2 \\
\Leftrightarrow & \neg\big((k_1 \vee p) \wedge (\neg p \vee k_2)\big) \vee k_1 \vee k_2 \\
\Leftrightarrow & \neg(k_1 \vee p) \vee \neg(\neg p \vee k_2) \vee k_1 \vee k_2 \\
\Leftrightarrow & (\neg k_1 \wedge \neg p) \vee (p \wedge \neg k_2) \vee k_1 \vee k_2 \\
\Leftrightarrow & (\neg k_1 \vee p \vee k_1 \vee k_2) \wedge (\neg k_1 \vee \neg k_2 \vee k_1 \vee k_2) \wedge (\neg p \vee p \vee k_1 \vee k_2) \wedge (\neg p \vee \neg k_2 \vee k_1 \vee k_2) \\
\Leftrightarrow & \top \wedge \top \wedge \top \wedge \top \\
\Leftrightarrow & \top
\end{array}$$

□

Definition 19 (Herleitungs-Begriff, $\vdash$) *Es sei M eine Menge von Klauseln und f sei eine einzelne Klausel. Die Formeln aus M bezeichnen wir als unsere Prämissen, die Formel f heißt Konklusion. Unser Ziel ist es, mit den Prämissen aus M die Konklusion f zu beweisen. Dazu definieren wir induktiv die Relation*

$M \vdash f$.

Wir lesen "$M \vdash f$" als "M leitet f her". Die induktive Definition ist wie folgt:

1. *Aus einer Menge M von Annahmen kann jede der Annahmen hergeleitet werden:*

   Falls $f \in M$ ist, dann gilt $M \vdash f$.

2. *Sind $k_1 \cup \{p\}$ und $\{\neg p\} \cup k_2$ Klauseln, die aus M hergeleitet werden können, so kann mit der Schnitt-Regel auch die Klausel $k_1 \cup k_2$ aus M hergeleitet werden:*

   Falls sowohl $M \vdash k_1 \cup \{p\}$ als auch $M \vdash \{\neg p\} \cup k_2$ gilt, dann gilt auch $M \vdash k_1 \cup k_2$. ◇

Beispiel: Um den Beweis-Begriff zu veranschaulichen geben wir ein Beispiel und zeigen

$$\{\ \{\neg p, q\},\ \{\neg q, \neg p\},\ \{\neg q, p\},\ \{q, p\}\ \} \vdash \bot.$$

Gleichzeitig zeigen wir anhand des Beispiels, wie wir Beweise zu Papier bringen:

1. Aus $\{\neg p, q\}$ und $\{\neg q, \neg p\}$ folgt mit der Schnitt-Regel $\{\neg p, \neg p\}$. Wegen $\{\neg p, \neg p\} = \{\neg p\}$ schreiben wir dies als

   $$\{\neg p, q\}, \{\neg q, \neg p\} \ \vdash\ \{\neg p\}.$$

   Bemerkung: Dieses Beispiel zeigt, dass die Klausel $k_1 \cup k_2$ durchaus auch weniger Elemente enthalten kann als die Summe $\texttt{card}(k_1) + \texttt{card}(k_2)$. Dieser Fall tritt genau dann ein, wenn es Literale gibt, die sowohl in k_1 als auch in k_2 vorkommen.

2. $\{\neg q, \neg p\},\ \{p, \neg q\} \ \vdash\ \{\neg q\}$.
3. $\{p, q\},\ \{\neg q\} \ \vdash\ \{p\}$.
4. $\{\neg p\},\ \{p\} \ \vdash\ \{\}$.

Als weiteres Beipiel zeigen wir nun, dass $p \to r$ aus $p \to q$ und $q \to r$ folgt. Dazu überführen wir zunächst alle Formeln in Klauseln:

$$\texttt{cnf}(p \to q) = \big\{\{\neg p, q\}\big\}, \quad \texttt{cnf}(q \to r) = \big\{\{\neg q, r\}\big\}, \quad \texttt{cnf}(p \to r) = \big\{\{\neg p, r\}\big\}.$$

Wir haben also $M = \{\ \{\neg p, q\},\ \{\neg q, r\}\ \}$ und müssen zeigen, dass

$$M \vdash \{\neg p, r\}$$

gilt. Der Beweis besteht aus einer einzigen Anwendung der Schnitt-Regel:

$$\{\neg p, q\},\ \{\neg q, r\} \ \vdash\ \{\neg p, r\}.$$

6.5.1 Eigenschaften des Herleitungs-Begriffs

Die Relation $\vdash$ hat zwei wichtige Eigenschaften:

Satz 20 (Korrektheit) *Ist $\{k_1, \cdots, k_n\}$ eine Menge von Klauseln und k eine einzelne Klausel, so haben wir:*

$$\text{Wenn } \{k_1, \cdots, k_n\} \vdash k \text{ gilt, dann gilt auch } \models k_1 \wedge \cdots \wedge k_n \to k.$$

Beweis: Der Beweis verläuft durch eine Induktion nach der Definition der Relation $\vdash$.

1. Fall: Es gilt $\{k_1, \cdots, k_n\} \vdash k$, weil $k \in \{k_1, \cdots, k_n\}$ ist. Dann gibt es also ein $i \in \{1, \cdots, n\}$, so dass $k = k_i$ ist. In diesem Fall müssen wir

   $$\models k_1 \wedge \cdots \wedge k_i \wedge \cdots \wedge k_n \to k_i$$

   zeigen, was offensichtlich ist.

2. Fall: Es gilt $\{k_1, \cdots, k_n\} \vdash k$, weil es eine aussagenlogische Variable p und Klauseln g und h gibt, so dass

   $$\{k_1, \cdots, k_n\} \vdash g \cup \{p\} \quad \text{und} \quad \{k_1, \cdots, k_n\} \vdash h \cup \{\neg p\}$$

   gilt und daraus haben wir mit der Schnitt-Regel auf

   $$\{k_1, \cdots, k_n\} \vdash g \cup h$$

   geschlossen, wobei $k = g \cup h$ gilt. Wir müssen nun zeigen, dass

   $$\models k_1 \wedge \cdots \wedge k_n \rightarrow g \vee h$$

   gilt. Es sei also $\mathcal{I}$ eine aussagenlogische Interpretation, so dass

   $$\mathcal{I}(k_1 \wedge \cdots \wedge k_n) = \texttt{True}$$

   ist. Dann müssen wir zeigen, dass

   $$\mathcal{I}(G) = \texttt{True} \quad \text{oder} \quad \mathcal{I}(H) = \texttt{True}$$

   ist. Nach Induktions-Voraussetzung wissen wir

   $$\models k_1 \wedge \cdots \wedge k_n \rightarrow g \vee p \quad \text{und} \quad \models k_1 \wedge \cdots \wedge k_n \rightarrow h \vee \neg p.$$

   Wegen $\mathcal{I}(k_1 \wedge \cdots \wedge k_n) = \texttt{True}$ folgt dann

   $$\mathcal{I}(g \vee p) = \texttt{True} \quad \text{und} \quad \mathcal{I}(h \vee \neg p) = \texttt{True}.$$

   Nun gibt es zwei Fälle:

   (a) Fall: $\mathcal{I}(p) = \texttt{True}$.
   Dann ist $\mathcal{I}(\neg p) = \texttt{False}$ und daher folgt aus der Tatsache, dass $\mathcal{I}(h \vee \neg p) = \texttt{True}$ ist, dass

   $$\mathcal{I}(h) = \texttt{True}$$

   sein muss. Daraus folgt aber sofort

   $$\mathcal{I}(g \vee h) = \texttt{True}. \checkmark$$

   (b) Fall: $\mathcal{I}(p) = \texttt{False}$.
   Nun folgt aus $\mathcal{I}(g \vee p) = \texttt{True}$, dass

   $$\mathcal{I}(g) = \texttt{True}$$

   gelten muss. Also gilt auch in diesem Fall

   $$\mathcal{I}(g \vee h) = \texttt{True}. \checkmark$$

□

Die Umkehrung dieses Satzes gilt leider nur in abgeschwächter Form und zwar dann, wenn k die leere Klausel ist, die dem Falsum entspricht.

Satz 21 (Widerlegungs-Vollständigkeit) *Ist $M = \{k_1, \cdots, k_n\}$ eine Menge von Klauseln, so haben wir:*
Wenn $\models k_1 \wedge \cdots \wedge k_n \rightarrow \bot$ gilt, dann gilt auch $M \vdash \{\}$.

Bemerkung: Es gibt alternative Definitionen des Herleitungs-Begriffs, die nicht nur widerlegungs-vollständig sondern tatsächlich vollständig sind, d.h. immer wenn $\models f_1 \wedge \cdots \wedge f_n \rightarrow g$ gilt, dann folgt auch

$$\{f_1, \cdots, f_n\} \vdash g.$$

Diese Herleitungs-Begriffe sind allerdings wesentlich komplexer und daher umständlicher zu implementieren. Wir werden später sehen, dass die Widerlegungs-Vollständigkeit für unsere Zwecke ausreichend ist. ◇

6.5.2 Beweis der Widerlegungs-Vollständigkeit

Der Beweis der Widerlegungs-Vollständigkeit der Aussagenlogik benötigt den Begriff der Erfüllbarkeit, den wir jetzt formal einführen.

Definition 22 (Erfüllbarkeit) *Es sei M eine Menge von aussagenlogischen Formeln. Falls es eine aussagenlogische Interpretation $\mathcal{I}$ gibt, die alle Formeln aus M erfüllt, für die also*

$$\mathcal{I}(f) = \texttt{True} \quad \text{für alle } f \in M$$

gilt, so nennen wir M erfüllbar.

Weiter sagen wir, dass M unerfüllbar ist und schreiben

$$M \models \bot,$$

wenn es keine aussagenlogische Interpretation $\mathcal{I}$ gibt, die gleichzeitig alle Formel aus M erfüllt. Bezeichnen wir die Menge der aussagenlogischen Interpretationen mit ALI*, so schreibt sich das formal als*

$$M \models \bot \quad \text{g.d.w.} \quad \forall \mathcal{I} \in \text{ALI} : \exists g \in M : \mathcal{I}(g) = \texttt{False}. \qquad \diamond$$

Bemerkung: Ist $M = \{f_1, \cdots, f_n\}$ eine Menge von aussagenlogischen Formeln, so können Sie sich leicht überlegen, dass M genau dann nicht erfüllbar ist, wenn

$$\models f_1 \wedge \cdots \wedge f_n \rightarrow \bot$$

gilt. $\diamond$

Wir führen den Beweis der Widerlegungs-Vollständigkeit mit Hilfe eines Programms, das in den Abbildungen 6.10, 6.11 und 6.12 auf den folgenden Seiten gezeigt ist. Sie finden dieses Programm unter der Adresse

`https://github.com/karlstroetmann/Logik/blob/master/Python/Completeness.ipynb`

im Netz. Die Grundidee bei diesem Programm besteht darin, dass wir versuchen, aus einer gegebenen Menge M von Klauseln alle Klauseln herzuleiten, die mit der Schnitt-Regel aus M herleitbar sind. Wenn wir dabei auch die leere Klausel herleiten, dann ist M aufgrund der Korrektheit der Schnitt-Regel offenbar unerfüllbar. Falls es uns aber nicht gelingt, die leere Klausel aus M abzuleiten, dann konstruieren wir aus der Menge aller Klauseln, die wir aus M hergeleitet haben, eine aussagenlogische Interpretation $\mathcal{I}$, die alle Klauseln aus M erfüllt. Damit haben wir dann die Unerfüllbarkeit der Menge M widerlegt. Wir diskutieren zunächst die Hilfsprozeduren, die in Abbildung 6.10 gezeigt sind.

1. Die Funktion `complement` erhält als Argument ein Literal l und berechnet das Komplement $\overline{l}$ dieses Literals. Falls das Literal l eine aussagenlogische Variable p ist, haben wir $\overline{p} = \neg p$. Falls l die Form $\neg p$ mit einer aussagenlogischen Variablen p hat, so gilt $\overline{\neg p} = p$.

2. Die Funktion `extractVariable` extrahiert die aussagenlogische Variable, die in einem Literal l enthalten ist. Die Implementierung verläuft analog zur Implementierung der Funktion `complement` über eine Fallunterscheidung, bei der wir berücksichtigen, dass l entweder die Form p oder die Form $\neg p$ hat, wobei p die zu extrahierende aussagenlogische Variable ist.

3. Die Funktion `collectVars` erhält als Argument eine Menge M von Klauseln, wobei die einzelnen Klauseln $C \in M$ als Mengen von Literalen dargestellt werden. Aufgabe der Funktion `collectVars` ist es, die Menge aller aussagenlogischen Variablen zu berechnen, die in einer der Klauseln C aus M vorkommen. Bei der Implementierung iterieren wir zunächst über die Klauseln C der Menge M und dann für jede Klausel C über die in C vorkommenden Literale l, wobei die Literale mit Hilfe der Funktion `extractVariable` in aussagenlogische Variablen umgewandelt werden.

4. Die Funktion `cutRule` erhält als Argumente zwei Klauseln C_1 und C_2 und berechnet die Menge aller Klauseln, die mit Hilfe einer Anwendung der Schnitt-Regel aus C_1 und C_2 gefolgert werden können. Beispielsweise können wir aus den beiden Klauseln

   $$\{p, q\} \quad \text{und} \quad \{\neg p, \neg q\}$$

   mit der Schnitt-Regel sowohl die Klausel

   $$\{q, \neg q\} \quad \text{als auch die Klausel} \quad \{p, \neg p\}$$

   herleiten.

```
1  def complement(l):
2      "Compute the complement of the literal l."
3      if isinstance(l, str):  # l is a propositional variable
4          return ('¬', l)
5      else:                   # l = ('¬', 'p')
6      return l[1]
7
8  def extractVariable(l):
9      "Extract the variable of the literal l."
10     if isinstance(l, str):  # l is a propositional variable
11         return l
12     else:                   # l = ('¬', 'p')
13         return l[1]
14
15 def collectVariables(M):
16     "Return the set of all variables occurring in M."
17     return { extractVariable(l) for C in M
18                                 for l in C
19            }
20
21 def cutRule(C1, C2):
22     "Return the set of all clauses that can be deduced by the cut rule from c1 and c2."
23     return { C1 - {l} | C2 - {complement(l)} for l in C1
24                                             if complement(l) in C2
25            }
```

Abbildung 6.10: Hilfsprozeduren, die in Abbildung 6.11 genutzt werden

Abbildung 6.11 zeigt die Funktion `saturate`. Diese Funktion erhält als Eingabe eine Menge `Clauses` von aussagenlogischen Klauseln, die als Mengen von Literalen dargestellt werden. Aufgabe der Funktion ist es, alle Klauseln herzuleiten, die mit Hilfe der Schnitt-Regel auf direktem oder indirekten Wege aus der Menge `Clauses` hergeleitet werden können. Genauer sagen wir, dass die Menge S der Klauseln, die von der Funktion `saturate` zurück gegeben wird, unter Anwendung der Schnitt-Regel saturiert ist, was formal wie folgt definiert ist:

1. Falls S die leere Klausel $\{\}$ enthält, dann ist S saturiert.
2. Andernfalls muss `Clauses` eine Teilmenge von S sein und es muss zusätzlich Folgendes gelten: Falls $C_1 \cup \{l\}$ und $C_2 \cup \{\overline{l}\}$ Klauseln aus S sind, dann ist auch die Klausel $C_1 \cup C_2$ ein Element der Klauselmenge S:

$$C_1 \cup \{l\} \in S \;\wedge\; C_2 \cup \{\overline{l}\} \in S \;\Rightarrow\; C_1 \cup C_2 \in S$$

Wir erläutern nun die Implementierung der Funktion `saturate`.

1. Die `while`-Schleife, die in Zeile 27 beginnt, hat die Aufgabe, die Schnitt-Regel so lange wie möglich anzuwenden, um mit Hilfe der Schnitt-Regel neue Klauseln aus den gegebenen Klauseln herzuleiten. Da die Bedingung dieser Schleife den Wert `True` hat, kann diese Schleife nur durch die Ausführung einer der beiden `return`-Befehle in Zeile 33 bzw. Zeile 36 abgebrochen werden.
2. In Zeile 28 wird die Menge `Derived` als die Menge der Klauseln definiert, die mit Hilfe der Schnitt-Regel aus zwei der Klauseln in der Menge `Clauses` gefolgert werden können.

```
26  def saturate(Clauses):
27      while True:
28          Derived = { C for C1 in Clauses
29                        for C2 in Clauses
30                        for C in cutRule(C1, C2)
31                    }
32          if frozenset() in Derived:
33              return { frozenset() }  # This is the set notation of ⊥.
34          Derived -= Clauses
35          if Derived == set():        # no new clauses found
36              return Clauses
37          Clauses |= Derived
```

Abbildung 6.11: Die Funktion `saturate`

3. Falls die Menge `Derived` die leere Klausel enthält, dann ist die Menge `Clauses` widersprüchlich und die Funktion `saturate` gibt als Ergebnis die Menge $\{\{\}\}$ zurück, wobei die innere Menge als `frozenset` dargestellt werden muss. Beachten Sie, dass die Menge $\{\{\}\}$ dem Falsum entspricht.

4. Andernfalls ziehen wir in Zeile 34 von der Menge `Derived` zunächst die Klauseln ab, die schon in der Menge `Clauses` vorhanden waren, denn es geht uns darum festzustellen, ob wir im letzten Schritt tatsächlich neue Klauseln gefunden haben, oder ob alle Klauseln, die wir im letzten Schritt in Zeile 28 hergeleitet haben, schon vorher bekannt waren.

5. Falls wir nun in Zeile 35 feststellen, dass wir keine neuen Klauseln hergeleitet haben, dann ist die Menge `Clauses` saturiert und wir geben diese Menge in Zeile 36 zurück.

6. Andernfalls fügen wir in Zeile 37 die Klauseln, die wir neu gefunden haben, zu der Menge `Clauses` hinzu und setzen die `while`-Schleife fort.

An dieser Stelle müssen wir uns überlegen, dass die `while`-Schleife tatsächlich irgendwann abbricht. Das hat zwei Gründe:

1. In jeder Iteration der Schleife wird die Anzahl der Elemente der Menge `Clauses` mindestens um Eins erhöht, denn wir wissen ja, dass die Menge `Derived`, die wir in Zeile 37 zur Menge `Clauses` hinzufügen, einerseits nicht leer ist und andererseits auch nur solche Klauseln enthält, die nicht bereits in `Clauses` auftreten.

2. Die Menge `Clauses`, mit der wir ursprünglich starten, enthält eine bestimmte Anzahl n von aussagenlogischen Variablen. Bei der Anwendung der Schnitt-Regel werden aber keine neue Variablen erzeugt. Daher bleibt die Anzahl der aussagenlogischen Variablen, die in `Clauses` auftreten, immer gleich. Damit ist natürlich auch die Anzahl der Literale, die in `Clauses` auftreten, beschränkt: Wenn es nur n aussagenlogische Variablen gibt, dann kann es auch höchstens $2 \cdot n$ verschiedene Literale geben. Jede Klausel aus `Clauses` ist aber eine Teilmenge der Menge aller Literale. Da eine Menge mit k Elementen insgesamt 2^k Teilmengen hat, gibt es höchstens $2^{2 \cdot n}$ verschiedene Klauseln, die in `Clauses` auftreten können.

Aus den beiden oben angegebenen Gründen können wir schließen, dass die `while`-Schleife in Zeile 20 spätestens nach $2^{2 \cdot n}$ Iterationen abgebrochen wird.

Als nächstes diskutieren wir die Implementierung der Funktion `findValuation`, die in Abbildung 6.12 gezeigt ist. Diese Funktion erhält als Eingabe eine Menge `Clauses` von Klauseln. Falls diese Menge widersprüchlich ist, soll die Funktion das Ergebnis `False` zurück geben. Andernfalls soll eine aussagenlogische Belegung $\mathcal{I}$ berechnet werden, unter der alle Klauseln aus der Menge `Clauses` erfüllt sind. Im Detail arbeitet die Funktion `findValuation` wie folgt.

```
38  def findValuation(Clauses):
39      "Given a set of Clauses, find an interpretation satisfying all of these clauses."
40      Variables = collectVariables(Clauses)
41      Clauses   = saturate(Clauses)
42      if frozenset() in Clauses:  # The set Clauses is inconsistent.
43          return False
44      Literals = set()
45      for p in Variables:
46          if any(C for C in Clauses
47                   if p in C and C - {p} <= { complement(l) for l in Literals }
48                ):
49              Literals |= { p }
50          else:
51              Literals |= { ('¬', p) }
52      return Literals
```

Abbildung 6.12: Die Funktion `findValuation`

1. Zunächst berechnen wir in Zeile 40 die Menge aller aussagenlogischen Variablen, die in der Menge `Clauses` auftreten. Wir benötigen diese Menge, denn in der aussagenlogischen Interpretation, die wir als Ergebnis zurück geben wollen, müssen wir diese Variablen auf die Menge $\{\texttt{True}, \texttt{False}\}$ abbilden.

2. In Zeile 41 saturieren wir die Menge `Clauses` und berechnen alle Klauseln, die aus der ursprünglich gegebenen Menge von Klauseln mit Hilfe der Schnitt-Regel hergeleitet werden können. Hier können zwei Fälle auftreten:

   (a) Falls die leere Klausel hergeleitet werden kann, dann folgt aus der Korrektheit der Schnitt-Regel, dass die ursprünglich gegebene Menge von Klauseln widersprüchlich ist und wir geben als Ergebnis an Stelle einer Belegung den Wert `False` zurück, denn eine widersprüchliche Menge von Klauseln ist sicher nicht erfüllbar.

   (b) Andernfalls berechnen wir nun eine aussagenlogische Belegung, unter der alle Klauseln aus der Menge `Clauses` wahr werden. Zu diesem Zweck berechnen wir zunächst eine Menge von Literalen, die wir in der Variablen `Literals` abspeichern. Die Idee ist dabei, dass wir die aussagenlogische Variable `p` genau dann in die Menge `Literals` aufnehmen, wenn die gesuchte Belegung $\mathcal{I}$ die aussagenlogische Variable `p` zu `True` auswertet. Andernfalls nehmen wir an Stelle von `p` das Literal $\neg\texttt{p}$ in der Menge `Literals` auf. Als Ergebnis geben wir daher in Zeile 52 die Menge `Literals` zurück. Die gesuchte aussagenlogische Belegung $\mathcal{I}$ kann dann gemäß der Formel

   $$\mathcal{I}(\texttt{p}) = \begin{cases} \texttt{True} & \text{falls } \texttt{p} \in \texttt{Literals} \\ \texttt{False} & \text{falls } \neg\texttt{p} \in \texttt{Literals} \end{cases}$$

   berechnet werden.

3. Die Berechnung der Menge `Literals` erfolgt nun über eine `for`-Schleife. Dabei ist der Gedanke, dass wir für eine aussagenlogische Variable `p` genau dann das Literal `p` zu der Menge `Literals` hinzufügen, wenn die Belegung $\mathcal{I}$ die Variable `p` auf `True` abbilden muss, um die Klauseln zu erfüllen. Andernfalls fügen wir stattdessen das Literal $\neg\texttt{p}$ zu dieser Menge hinzu.

   Die Bedingung dafür ist wie folgt: Angenommen, wir haben bereits Werte für die Variablen $p_1, \cdots, p_n$ in der Menge `Literals` gefunden. Die Werte dieser Variablen seien durch die Literale $l_1, \cdots, l_n$ in der Menge `Literals` wie folgt festgelegt: Wenn $l_i = p_i$ ist, dann gilt $\mathcal{I}(p_i) = \texttt{True}$ und falls $l_i = \neg p_i$ gilt, so haben wir $\mathcal{I}(p_i) = \texttt{False}$. Nehmen wir nun weiter an, dass eine Klausel C in der Menge `Clauses` existiert, so dass

   $$C \backslash \{p\} \subseteq \{\overline{l_1}, \cdots, \overline{l_n}\} \quad \text{und} \quad \texttt{p} \in C$$

gilt. Wenn $\mathcal{I}(C) = \texttt{True}$ gelten soll, dann muss $\mathcal{I}(\texttt{p}) = \texttt{True}$ gelten, denn nach Konstruktion von $\mathcal{I}$ gilt

$$\mathcal{I}(\overline{l_i}) = \texttt{False} \quad \text{für alle } i \in \{1, \cdots, n\}$$

und damit ist p das einzige Literal in der Klausel C, das wir mit Hilfe der Belegung $\mathcal{I}$ überhaupt noch wahr machen können. In diesem Fall fügen wir also das Literal p in die Menge Literals ein.

Der entscheidende Punkt ist nun der Nachweis, dass die Funktion findValuation in dem Falle, dass in Zeile 43 nicht der Wert False zurück gegeben wird, eine aussagenlogische Belegung $\mathcal{I}$ berechnet, bei der alle Klauseln aus der Menge Clauses den Wert True erhalten. Um diesen Nachweis zu erbringen, nummerieren wie die aussagenlogischen Variablen, die in der Menge Clauses auftreten, in derselben Reihenfolge durch, in der diese Variablen in der for-Schleife in Zeile 45 betrachtet werden. Wir bezeichnen diese Variablen als

$$p_1, p_2, p_3, \cdots, p_k$$

und zeigen durch Induktion nach n, dass nach n Durchläufen der Schleife für jede Klausel $D \in \texttt{Clauses}$, in der nur die Variablen $p_1, \cdots, p_n$ vorkommen,

$$\mathcal{I}(D) = \texttt{True}$$

gilt.

I.A.: $n = 1$.

In diesem Fall muss entweder

$$D = \{p\} \quad \text{oder} \quad D = \{\neg p\}$$

gelten. An dieser Stelle brauchen wir eine Fallunterscheidung.

(a) $D = \{p\}$.
Daraus folgt aber sofort

$$D \backslash \{p\} = \{\} \subseteq \{\overline{l} \mid l \in \texttt{Literals}\}.$$

Also ist die Bedingung in Zeile 47 erfüllt und wir haben $\texttt{p} \in \texttt{Literals}$. Damit gilt $\mathcal{I}(p) = \texttt{True}$ nach Definition von $\mathcal{I}$.

(b) $D = \{\neg p\}$.
Würde es jetzt eine Klausel $E = \{p\} \in \texttt{Clauses}$ geben, so könnten wir aus den beiden Klauseln D und E sofort die leere Klausel $\{\}$ herleiten und die Funktion findValuation würde in Zeile 43 den Wert False zurück geben. Da wir aber vorausgesetzt haben, dass dies nicht passiert, kann es keine solche Klausel E geben. Damit ist die Bedingung in Zeile 46 falsch und folglich gilt $\neg p \in \texttt{Literals}$. Nach Definition von $\mathcal{I}$ folgt dann $\mathcal{I}(\neg p) = \texttt{True}$.

Damit haben wir in jedem Fall $\mathcal{I}(D) = \texttt{True}$.

I.S.: $n \mapsto n + 1$.

Wir setzen nun voraus, dass die Behauptung vor dem $(n+1)$-ten Durchlauf der for-Schleife gilt und haben zu zeigen, dass die Behauptung dann auch nach diesem Durchlauf erfüllt ist. Sei dazu D eine Klausel, in der nur die Variablen $p_1, \cdots, p_n, p_{n+1}$ vorkommen. Die Klausel ist dann eine Teilmenge einer Menge der Form

$$\{l_1, \cdots, l_n, l_{n+1}\}, \quad \text{wobei } l_i \in \{p_i, \neg p_i\} \text{ für alle } i \in \{1, \cdots, n+1\} \text{ gilt.}$$

Nun gibt es mehrere Möglichkeiten, die wir getrennt untersuchen.

(a) Es gibt ein $i \in \{1, \cdots, n\}$, so dass $l_i \in D$ und $\mathcal{I}(l_i) = \texttt{True}$ ist.
Da eine Klausel als Disjunktion ihrer Literale aufgefasst wird, gilt dann auch $\mathcal{I}(D) = \texttt{True}$ unabhängig davon, ob $\mathcal{I}(p_{n+1})$ den Wert True oder False hat.

(b) Für alle $i \in \{1, \cdots, n\}$ mit $l_i \in D$ gilt $\mathcal{I}(l_i) = \texttt{False}$ und es gilt $l_{n+1} = p_{n+1}$.
Dann gilt für die Klausel D gerade die Bedingung

$$C \backslash \{p_{n+1}\} \subseteq \{\overline{l_1}, \cdots, \overline{l_n}\} \quad \text{und} \quad p_{n+1} \in C$$

und daher wird in Zeile 44 der Funktion `findValuation` das Literal p_{n+1} zu der Menge `Literals` hinzugefügt. Nach Definition der Belegung $\mathcal{I}$, die von der Funktion `findValuation` zurück gegeben wird, heißt dies gerade, dass

$$\mathcal{I}(p_{n+1}) = \texttt{True}$$

ist und dann gilt natürlich auch $\mathcal{I}(D) = \texttt{True}$.

(c) Für alle $i \in \{1, \cdots, n\}$ mit $l_i \in D$ gilt $\mathcal{I}(l_i) = \texttt{False}$ und es gilt $l_{n+1} = \neg p_{n+1}$.
An dieser Stelle ist eine weitere Fall-Unterscheidung notwendig.

i. Es gibt eine weitere Klausel C in der Menge `Clauses`, so dass

$$C \backslash \{p_{n+1}\} \subseteq \{\overline{l_1}, \cdots, \overline{l_n}\} \quad \text{und} \quad p_{n+1} \in C$$

gilt. Hier sieht es zunächst so aus, als ob wir ein Problem hätten, denn in diesem Fall würde um die Klausel C wahr zu machen das Literal p_{n+1} zur Menge `Literals` hinzugefügt und damit wäre zunächst $\mathcal{I}(p_{n+1}) = \texttt{True}$ und damit $\mathcal{I}(\neg p_{n+1}) = \texttt{False}$, woraus insgesamt $\mathcal{I}(D) = \texttt{False}$ folgern würde. In diesem Fall würden sich die Klauseln C und D in der Form

$$C = C' \cup \{p_{n+1}\}, \quad D = D' \cup \{\neg p_{n+1}\}$$

schreiben lassen, wobei

$$C' \subseteq \left\{\overline{l} \mid l \in \texttt{Literals}\right\} \quad \text{und} \quad D' \subseteq \left\{\overline{l} \mid l \in \texttt{Literals}\right\}$$

gelten würde. Daraus würde sowohl

$$\mathcal{I}(C') = \texttt{False} \quad \text{als auch} \quad \mathcal{I}(D') = \texttt{False}$$

folgen und das würde auch

$$\mathcal{I}(C' \cup D') = \texttt{False} \tag{$*$}$$

implizieren. Die entscheidende Beobachtung ist nun, dass die Klausel $C' \cup D'$ mit Hilfe der Schnitt-Regel aus den beiden Klauseln

$$C = C' \cup \{p_{n+1}\}, \quad D = D' \cup \{\neg p_{n+1}\},$$

gefolgert werden kann. Das heißt dann aber, dass die Klausel $C' \cup D'$ ein Element der Menge `Clauses` sein muss, denn die Menge `Clauses` ist ja saturiert! Da die Klausel $C' \cup D'$ außerdem nur die aussagenlogischen Variablen $p_1, \cdots, p_n$ enthält, gilt nach Induktions-Voraussetzung

$$\mathcal{I}(C' \cup D') = \texttt{True}.$$

Dies steht aber im Widerspruch zu $(*)$. Dieser Widerspruch zeigt, dass es keine Klausel $C \in$ `Clauses` mit

$$C \subseteq \{\overline{l} \mid l \in \texttt{Literals}\} \cup \{p_{n+1}\} \quad \text{und} \quad p_{n+1} \in C$$

geben kann und damit tritt der hier untersuchte Fall gar nicht auf.

ii. Es gibt <u>keine</u> Klausel C in der Menge `Clauses`, so dass

$$C \subseteq \{\overline{l} \mid l \in \texttt{Literals}\} \cup \{p_{n+1}\} \quad \text{und} \quad p_{n+1} \in C$$

gilt. In diesem Fall wird das Literal $\neg p_{n+1}$ zur Menge `Literals` hinzugefügt und damit gilt zunächst $\mathcal{I}(p_{n+1}) = \texttt{False}$ und folglich $\mathcal{I}(\neg p_{n+1}) = \texttt{True}$, woraus schließlich $\mathcal{I}(D) = \texttt{True}$ folgt.

Wir sehen, dass der erste Fall der vorherigen Fall-Unterscheidung nicht auftritt und dass im zweiten Fall $\mathcal{I}(D) = \texttt{True}$ gilt, womit wir insgesamt $\mathcal{I}(D) = \texttt{True}$ gezeigt haben. Damit ist der Induktions-Schritt abgeschlossen.

Da jede Klausel $C \in$ `Clauses` nur eine endliche Anzahl von Variablen enthält, haben wir insgesamt gezeigt, dass für alle diese Klauseln $\mathcal{I}(C) = \texttt{True}$ gilt. □

Beweis der Widerlegungs-Vollständigkeit der Schnitt-Regel: Wir haben nun alles Material zusammen um zeigen zu können, dass die Schnitt-Regel widerlegungs-vollständig ist. Wir nehmen also an, dass M eine endliche Menge von Klauseln ist, die nicht erfüllbar ist, was wir als

$$M \models \bot$$

schreiben. Wir rufen die Funktion `findValuation` mit dieser Menge M als Argument auf. Jetzt gibt es zwei Möglichkeiten:

1. Fall: Die Funktion `findValuation` liefert als Ergebnis `False`. Nach Konstruktion der Funktionen `findValuation` `saturate` tritt dieser Fall nur ein, wenn sich die leere Klausel $\{\}$ aus den Klauseln der Menge M mit Hilfe der Schnitt-Regel herleiten lässt. Dann haben wir also

   $$M \vdash \{\},$$

   was zu zeigen war.

2. Fall: Die Funktion `findValuation` liefert als Ergebnis eine aussagenlogische Belegung $\mathcal{I}$. Bei der Diskussion der Funktion `findValuation` haben wir gezeigt, dass für alle Klauseln $D \in$ `Clauses`

   $$\mathcal{I}(D) = \texttt{True}$$

   gilt. Die Menge M ist aber eine Teilmenge der Menge `Clauses` und damit sehen wir, dass die Menge M erfüllbar ist. Dies steht im Widerspruch zu $M \models \bot$ und folglich kann der zweite Fall nicht auftreten.

Folglich liefert die Funktion `findValuation` für eine unerfüllbare Menge von Klauseln immer das Ergebnis `False`, was impliziert, dass $M \vdash \{\}$ gilt. □

6.6 Das Verfahren von Davis und Putnam

In der Praxis stellt sich oft die Aufgabe, für eine gegebene Menge von Klauseln K eine aussagenlogische Belegung $\mathcal{I}$ zu berechnen, so dass

$$\texttt{evaluate}(C, \mathcal{I}) = \texttt{True} \quad \text{für alle } C \in K$$

gilt. In diesem Fall sagen wir auch, dass die Belegung $\mathcal{I}$ eine Lösung der Klausel-Menge K ist. Im letzten Abschnitt haben wir bereits die Funktion `findValuation` kennengelernt, mit der wir eine solche Belegung berechnen könnten. Bedauerlicherweise ist diese Funktion für eine praktische Anwendung nicht effizient genug. Wir werden daher in diesem Abschnitt ein Verfahren vorstellen, mit dem die Berechnung einer Lösung einer aussagenlogischen Klausel-Menge in vielen praktisch relevanten Fällen auch dann möglich ist, wenn die Menge der Klauseln K groß ist. Dieses Verfahren geht auf Davis und Putnam [DP60, DLL62] zurück. Verfeinerungen dieses Verfahrens werden beispielsweise eingesetzt, um die Korrektheit digitaler elektronischer Schaltungen nachzuweisen.

Um das Verfahren zu motivieren, überlegen wir zunächst, bei welcher Form der Klausel-Menge K unmittelbar klar ist, ob es eine Belegung gibt, die K löst und wie diese Belegung aussieht. Betrachten wir dazu ein Beispiel:

$$K_1 = \{\ \{p\},\ \{\neg q\},\ \{r\},\ \{\neg s\},\ \{\neg t\}\ \}$$

Die Klausel-Menge K_1 entspricht der aussagenlogischen Formel

$$p \wedge \neg q \wedge r \wedge \neg s \wedge \neg t.$$

Daher ist K_1 lösbar und die Belegung

$$\mathcal{I} = \{\ \langle p, \texttt{True}\rangle,\ \langle q, \texttt{False}\rangle,\ \langle r, \texttt{True}\rangle,\ \langle s, \texttt{False}\rangle,\ \langle t, \texttt{False}\rangle\ \}$$

ist eine Lösung. Betrachten wir eine weiteres Beispiel:

$$K_2 = \{\ \{\}, \{p\},\ \{\neg q\},\ \{r\}\ \}$$

Diese Klausel-Menge entspricht der Formel

$$\bot \wedge p \wedge \neg q \wedge r.$$

Offensichtlich ist K_2 unlösbar. Als letztes Beispiel betrachten wir

$$K_3 = \big\{\{p\}, \{\neg q\}, \{\neg p\}\big\}.$$

Diese Klausel-Menge kodiert die Formel

$$p \wedge \neg q \wedge \neg p$$

und ist offenbar ebenfalls unlösbar, denn eine Lösung $\mathcal{I}$ müsste die aussagenlogische Variable p gleichzeitig wahr und falsch machen. Wir nehmen die an den letzten drei Beispielen gemachten Beobachtungen zum Anlass für zwei Definitionen.

Definition 23 (Unit-Klausel) *Eine Klausel C heißt Unit-Klausel, wenn C nur aus einem Literal besteht. Es gilt dann entweder*

$$C = \{p\} \quad \textit{oder} \quad C = \{\neg p\}$$

für eine geeignete Aussage-Variable p. ◇

Definition 24 (Triviale Klausel-Mengen) *Eine Klausel-Menge K heißt trivial wenn einer der beiden folgenden Fälle vorliegt:*

1. *K enthält die leere Klausel, es gilt also $\{\} \in K$.*

   In diesem Fall ist K offensichtlich unlösbar.

2. *K enthält nur Unit-Klauseln mit verschiedenen Aussage-Variablen. d.h. es kann nicht sein, dass es eine aussagenlogische Variable p gibt, so dass K sowohl die Klausel $\{p\}$, als auch die Klausel $\{\neg p\}$ enthält. Bezeichnen wir die Menge der aussagenlogischen Variablen mit $\mathcal{P}$, so schreibt sich diese Bedingung als*

   $$\big(\forall C \in K : \texttt{card}(C) = 1\big) \;\wedge\; \forall p \in \mathcal{P} : \neg\big(\{p\} \in K \wedge \{\neg p\} \in K\big).$$

   In diesem Fall können wir die aussagenlogische Belegung $\mathcal{I}$ wie folgt definieren:

   $$\mathcal{I}(p) = \begin{cases} \texttt{True} & \textit{falls} \quad \{p\} \in K, \\ \texttt{False} & \textit{falls} \quad \{\neg p\} \in K. \end{cases}$$

   eine Lösung der Klausel-Menge K. ◇

Bemerkung: Beachten Sie, dass wir in dieser Vorlesung das Wort trivial an zwei verschiedenen Stellen in unterschiedlicher Bedeutung benutzen:

1. Einerseits haben wir den Begriff der trivialen Klausel definiert: Eine Klausel ist genau dann trivial, wenn sie eine Tautologie ist.
2. Andererseits haben wir gerade den Begriff der trivialen Klausel-Menge definiert. ◇

Wie können wir nun eine Menge von Klauseln so vereinfachen, dass die Menge schließlich nur noch aus Unit-Klauseln besteht? Es gibt drei Möglichkeiten, Klauselmengen zu vereinfachen:

1. Schnitt-Regel,
2. Subsumption und
3. Fallunterscheidung.

Wir betrachten diese Möglickeiten jetzt der Reihe nach.

6.6.1 Vereinfachung mit der Schnitt-Regel

Eine typische Anwendung der Schnitt-Regel hat die Form:

$$\frac{C_1 \cup \{p\} \quad \{\neg p\} \cup C_2}{C_1 \cup C_2}$$

Die hierbei erzeugte Klausel $C_1 \cup C_2$ wird in der Regel mehr Literale enthalten als die Prämissen $C_1 \cup \{p\}$ und $\{\neg p\} \cup C_2$. Enthält die Klausel $C_1 \cup \{p\}$ insgesamt $m+1$ Literale und enthält die Klausel $\{\neg p\} \cup C_2$ insgesamt $n+1$ Literale, so kann die Konklusion $C_1 \cup C_2$ bis zu $m+n$ Literale enthalten. Natürlich können es auch weniger Literale sein, und zwar dann, wenn es Literale gibt, die sowohl in C_1 als auch in C_2 auftreten. Oft ist $m+n$ größer als $m+1$ und als $n+1$. Die Klauseln wachsen nur dann sicher nicht, wenn $n = 0$ oder $m = 0$ ist. Dieser Fall liegt vor, wenn einer der beiden Klauseln nur aus einem Literal besteht und folglich eine Unit-Klausel ist. Da es unser Ziel ist, die Klausel-Mengen zu vereinfachen, lassen wir nur solche Anwendungen der Schnitt-Regel zu, bei denen eine der Klausel eine Unit-Klausel ist. Solche Schnitte bezeichnen wir als Unit-Schnitte. Um alle mit einer gegebenen Unit-Klausel $\{l\}$ möglichen Schnitte durchführen zu können, definieren wir eine Funktion

$$\texttt{unitCut} : 2^{\mathcal{K}} \times \mathcal{L} \to 2^{\mathcal{K}}$$

so, dass für eine Klausel-Menge K und ein Literal l die Funktion $\texttt{unitCut}(K, l)$ die Klausel-Menge K soweit wie möglich mit Unit-Schnitten mit der Klausel $\{l\}$ vereinfacht:

$$\texttt{unitCut}(K, l) = \Big\{ C \backslash \{\overline{l}\} \;\Big|\; C \in K \Big\}.$$

Beachten Sie, dass die Menge $\texttt{unitCut}(K, l)$ genauso viele Klauseln enthält wie die Menge K. Allerdings sind die Klauseln aus der Menge K, die das Literal $\overline{l}$ enthalten, verkleinert worden. Alle anderen Klauseln aus K bleiben unverändert.

6.6.2 Vereinfachung durch Subsumption

Das Prinzip der Subsumption demonstrieren wir zunächst an einem Beispiel. Wir betrachten

$$K = \big\{\{p, q, \neg r\}, \{p\}\big\} \cup M.$$

Offenbar impliziert die Klausel $\{p\}$ die Klausel $\{p, q, \neg r\}$, denn immer wenn $\{p\}$ erfüllt ist, ist automatisch auch $\{q, p, \neg r\}$ erfüllt. Das liegt daran, dass

$$\models p \to q \vee p \vee \neg r$$

gilt. Allgemein sagen wir, dass eine Klausel C von einer Unit-Klausel U subsumiert wird, wenn

$$U \subseteq C$$

gilt. Ist K eine Klausel-Menge mit $C \in K$ und $U \in K$ und wird C durch U subsumiert, so können wir die Menge K durch Unit-Subsumption zu der Menge $K - \{C\}$ verkleinern, wir können also die Klausel C aus K löschen. Dazu definieren wir eine Funktion

$$\texttt{subsume} : 2^{\mathcal{K}} \times \mathcal{L} \to 2^{\mathcal{K}}$$

die eine gegebene Klauselmenge K, welche die Unit-Klausel $\{l\}$ enthält, mittels Subsumption dadurch vereinfacht, dass alle durch $\{l\}$ subsumierten Klauseln aus K gelöscht werden. Die Unit-Klausel $\{l\}$ selbst behalten wir natürlich. Daher definieren wir:

$$\texttt{subsume}(K, l) := \big(K \backslash \{C \in K \mid l \in C\}\big) \cup \big\{\{l\}\big\} = \big\{C \in K \mid l \notin C\big\} \cup \big\{\{l\}\big\}.$$

In der obigen Definition muss $\{l\}$ in das Ergebnis eingefügt werden, weil die Menge $\big\{C \in K \mid l \notin C\big\}$ die Unit-Klausel $\{l\}$ nicht enthält. Die beiden Klausel-Mengen $\texttt{subsume}(K, l)$ und K sind genau dann äquivalent, wenn $\{l\} \in K$ gilt.

6.6.3 Vereinfachung durch Fallunterscheidung

Ein Kalkül, der nur mit Unit-Schnitten und Subsumption arbeitet, ist nicht widerlegungs-vollständig. Wir brauchen daher eine weitere Möglichkeit, Klausel-Mengen zu vereinfachen. Eine solche Möglichkeit bietet das Prinzip der Fallunterscheidung. Dieses Prinzip basiert auf dem folgenden Satz.

Satz 25 *Ist K eine Menge von Klauseln und ist p eine aussagenlogische Variable, so ist K genau dann erfüllbar, wenn $K \cup \big\{\{p\}\big\}$ oder $K \cup \big\{\{\neg p\}\big\}$ erfüllbar ist.*

Beweis: Ist K erfüllbar durch eine Belegung $\mathcal{I}$, so gibt es für $\mathcal{I}(p)$ zwei Möglichkeiten: Falls $\mathcal{I}(p) = \texttt{True}$ ist, ist damit auch die Menge $K \cup \big\{\{p\}\big\}$ erfüllbar, andernfalls ist $K \cup \big\{\{\neg p\}\big\}$ erfüllbar.

Da K sowohl eine Teilmenge von $K \cup \big\{\{p\}\big\}$ als auch von $K \cup \big\{\{\neg p\}\big\}$ ist, ist klar, dass K erfüllbar ist, wenn eine dieser Mengen erfüllbar sind. □

Wir können nun eine Menge K von Klauseln dadurch vereinfachen, dass wir eine aussagenlogische Variable p wählen, die in K vorkommt. Anschließend bilden wir die Mengen

$$K_1 := K \cup \big\{\{p\}\big\} \quad \text{und} \quad K_2 := K \cup \big\{\{\neg p\}\big\}$$

und untersuchen rekursiv ob K_1 erfüllbar ist. Falls wir eine Lösung für K_1 finden, ist dies auch eine Lösung für die ursprüngliche Klausel-Menge K und wir haben unser Ziel erreicht. Andernfalls untersuchen wir rekursiv ob K_2 erfüllbar ist. Falls wir nun eine Lösung finden, ist dies auch eine Lösung von K und wenn wir weder für K_1 noch für K_2 eine Lösung finden, dann kann auch K keine Lösung haben, denn jede Lösung $\mathcal{I}$ von K muss die Variable p entweder wahr oder falsch machen. Die rekursive Untersuchung von K_1 bzw. K_2 ist leichter als die Untersuchung von K, weil wir ja in K_1 und K_2 mit den Unit-Klausel $\{p\}$ bzw. $\{\neg p\}$ sowohl Unit-Subsumptionen als auch Unit-Schnitte durchführen können.

6.6.4 Der Algorithmus

Wir können jetzt den Algorithmus von Davis und Putnam skizzieren. Gegeben sei eine Menge K von Klauseln. Gesucht ist dann eine Lösung von K. Wir suchen also eine Belegung $\mathcal{I}$, so dass gilt:

$$\mathcal{I}(C) = \texttt{True} \quad \text{für alle } C \in K.$$

Das Verfahren von Davis und Putnam besteht nun aus den folgenden Schritten.

1. Führe alle Unit-Schnitte und Unit-Subsumptionen aus, die mit Klauseln aus K möglich sind.
2. Falls K nun trivial ist, sind wir fertig.
3. Andernfalls wählen wir eine aussagenlogische Variable p, die in K auftritt.
   (a) Jetzt versuchen wir rekursiv die Klausel-Menge
   $$K \cup \big\{\{p\}\big\}$$
   zu lösen. Falls diese gelingt, haben wir eine Lösung von K.
   (b) Andernfalls versuchen wir die Klausel-Menge
   $$K \cup \big\{\{\neg p\}\big\}$$
   zu lösen. Wenn auch dies fehlschlägt, ist K unlösbar, andernfalls haben wir eine Lösung von K.

Für die Implementierung ist es zweckmäßig, die beiden oben definierten Funktionen `unitCut()` und `subsume()` zu einer Funktion zusammen zu fassen. Wir definieren daher die Funktion

$$\texttt{reduce} : 2^{\mathcal{K}} \times \mathcal{L} \to 2^{\mathcal{K}}$$

wie folgt:

$$\texttt{reduce}(K, l) = \Big\{ C \backslash \{\overline{l}\} \mid C \in K \wedge \overline{l} \in C \Big\} \cup \Big\{ C \in K \mid \overline{l} \notin C \wedge l \notin C \Big\} \cup \big\{\{l\}\big\}.$$

Die Menge enthält also einerseits die Ergebnisse von Schnitten mit der Unit-Klausel $\{l\}$ und andererseits nur die Klauseln C, die mit l nichts zu tun haben, weil weder $l \in C$ noch $\overline{l} \in C$ gilt. Außerdem fügen wir noch die Unit-Klausel $\{l\}$ hinzu. Dadurch erreichen wir, dass die beiden Mengen K und $\texttt{reduce}(K, l)$ logisch äquivalent sind, falls $\{l\} \in K$ gilt.

6.6.5 Ein Beispiel

Zur Veranschaulichung demonstrieren wir das Verfahren von Davis und Putnam an einem Beispiel. Die Menge K sei wie folgt definiert:

$$K := \Big\{ \{p,q,s\},\ \{\neg p,r,\neg t\},\ \{r,s\},\ \{\neg r,q,\neg p\},\{\neg s,p\},\ \{\neg p,\neg q,s,\neg r\},\ \{p,\neg q,s\},\ \{\neg r,\neg s\},\ \{\neg p,\neg s\} \Big\}.$$

Wir zeigen nun mit dem Verfahren von Davis und Putnam, dass K nicht lösbar ist. Da die Menge K keine Unit-Klauseln enthält, ist im ersten Schritt nichts zu tun. Da K nicht trivial ist, sind wir noch nicht fertig. Also gehen wir jetzt zu Schritt 3 und wählen eine aussagenlogische Variable, die in K auftritt. An dieser Stelle ist es sinnvoll eine Variable zu wählen, die in möglichst vielen Klauseln von K auftritt. Wir wählen daher die aussagenlogische Variable p.

1. Zunächst bilden wir die Menge

   $$K_0 := K \cup \big\{\{p\}\big\}$$

   und versuchen, diese Menge zu lösen. Dazu bilden wir

   $$K_1 := \texttt{reduce}\big(K_0, p\big) = \Big\{ \{r,\neg t\},\ \{r,s\},\ \{\neg r,q\},\ \{\neg q,s,\neg r\},\ \{\neg r,\neg s\},\ \{\neg s\},\ \{p\} \Big\}.$$

   Die Klausel-Menge K_1 enthält die Unit-Klausel $\{\neg s\}$, so dass wir als nächstes mit dieser Klausel reduzieren können:

   $$K_2 := \texttt{reduce}\big(K_1, \neg s\big) = \Big\{ \{r,\neg t\},\ \{r\},\ \{\neg r,q\},\ \{\neg q,\neg r\},\ \{\neg s\},\ \{p\} \Big\}.$$

   Hier haben wir nun die neue Unit-Klausel $\{r\}$, mit der wir weiter reduzieren:

   $$K_3 := \mathit{reduce}\big(K_2, r\big) = \Big\{ \{r\},\ \{q\},\ \{\neg q\},\ \{\neg s\},\ \{p\} \Big\}$$

   Da K_3 die Unit-Klausel $\{q\}$ enthält, reduzieren wir jetzt mit q:

   $$K_4 := \mathit{reduce}\big(K_2, q\big) = \Big\{ \{r\},\ \{q\},\ \{\},\ \{\neg s\},\ \{p\} \Big\}.$$

   Die Klausel-Menge K_4 enthält die leere Klausel und ist damit unlösbar.

2. Also bilden wir jetzt die Menge

   $$K_5 := K \cup \big\{\{\neg p\}\big\}$$

   und versuchen, diese Menge zu lösen. Dazu bilden wir

   $$K_6 = \mathit{reduce}\big(K_5, \neg p\big) = \Big\{ \{q,s\},\ \{r,s\},\ \{\neg s\},\ \{\neg q,s\},\ \{\neg r,\neg s\},\ \{\neg p\} \Big\}.$$

   Die Menge K_6 enthält die Unit-Klausel $\{\neg s\}$. Wir bilden daher

   $$K_7 = \mathit{reduce}\big(K_6, \neg s\big) = \Big\{ \{q\},\ \{r\},\ \{\neg s\},\ \{\neg q\},\ \{\neg p\} \Big\}.$$

   Die Menge K_7 enthält die neue Unit-Klausel $\{q\}$, mit der wir als nächstes reduzieren:

   $$K_8 = \mathit{reduce}\big(K_7, q\big) = \Big\{ \{q\},\ \{r\},\ \{\neg s\},\ \{\},\ \{\neg p\} \Big\}.$$

   Da K_8 die leere Klausel enthält, ist K_8 und damit auch die ursprünglich gegebene Menge K unlösbar.

Bei diesem Beispiel hatten wir Glück, denn wir mussten nur eine einzige Fallunterscheidung durchführen. Bei

komplexeren Beispielen ist es häufig so, dass wir innerhalb einer Fallunterscheidung eine weitere Fallunterscheidungen durchführen müssen.

6.6.6 Implementierung des Algorithmus von Davis und Putnam

Wir zeigen jetzt die Implementierung der Funktion `solve`, mit der die Frage, ob eine Menge von Klauseln erfüllbar ist, beantwortet werden kann. Die Implementierung ist in Abbildung 6.13 auf Seite 101 gezeigt. Die Funktion erhält zwei Argumente: Die Mengen `Clauses` und `Literals`. Hier ist `Clauses` eine Menge von Klauseln und `Literals` ist eine Menge von Literalen. Falls die Menge `Clauses` erfüllbar ist, so liefert der Aufruf

```
solve(Clauses, Literals)
```

eine Menge von Unit-Klauseln `Result`, so dass jede Belegung $\mathcal{I}$, die alle Unit-Klauseln aus `Result` erfüllt, auch alle Klauseln aus der Menge `Clauses` erfüllt. Falls die Menge `Clauses` nicht erfüllbar ist, liefert der Aufruf

```
solve(Clauses, Literals)
```

als Ergebnis die Menge $\big\{\{\}\big\}$ zurück, denn die leere Klausel repräsentiert die unerfüllbare Formel $\bot$.

Sie fragen sich vielleicht, wozu wir in der Funktion `solve` die Menge `Literals` brauchen. Der Grund ist, dass wir uns bei den rekursiven Aufrufen merken müssen, welche Literale wir schon benutzt haben. Diese Literale sammeln wir in der Menge `Literals`.

```
1  def solve(Clauses, Literals):
2      S      = saturate(Clauses);
3      empty  = frozenset()
4      Falsum = {empty}
5      if empty in S:
6          return Falsum
7      if all(len(C) == 1 for C in S):
8          return S
9      l      = selectLiteral(S, Literals)
10     negL   = complement(l)
11     Result = solve(S | { frozenset({l}) }, Literals | { l })
12     if Result != Falsum:
13         return Result
14     return solve(S | { frozenset({negL}) }, Literals | { l })
```

Abbildung 6.13: Die Funktion `solve`

Die in Abbildung 6.13 gezeigte Implementierung funktioniert wie folgt:

1. In Zeile 2 reduzieren wir mit Hilfe der Methode `saturate` solange wie möglich die gegebene Klausel-Menge `Clauses` mit Hilfe von Unit-Schnitten und entfernen alle Klauseln, die durch Unit-Klauseln subsumiert werden.

2. Anschließend testen wir in Zeile 5, ob die so vereinfachte Klausel-Menge `S` die leere Klausel enthält und geben in diesem Fall als Ergebnis die Menge $\big\{\{\}\big\}$ zurück.

3. Dann testen wir in Zeile 7, ob bereits alle Klauseln C aus der Menge `S` Unit-Klauseln sind. Wenn dies so ist, dann ist die Menge `S` trivial und wir geben diese Menge als Ergebnis zurück.

4. Andernfalls wählen wir in Zeile 9 ein Literal `l` aus der Menge `S`, dass wir noch nicht benutzt haben. Wir untersuchen dann in Zeile 11 rekursiv, ob die Menge

   $$\texttt{S} \cup \big\{\{\texttt{l}\}\big\}$$

   lösbar ist. Dabei gibt es zwei Fälle:

(a) Falls diese Menge lösbar ist, geben wir die Lösung dieser Menge als Ergebnis zurück.

(b) Sonst prüfen wir rekursiv, ob die Menge

$$\texttt{S} \cup \Big\{ \big\{ \overline{\texttt{l}} \big\} \Big\}$$

lösbar ist. Ist diese Menge lösbar, so ist diese Lösung auch eine Lösung der Menge `Clauses` und wir geben diese Lösung zurück. Ist die Menge unlösbar, dann muss auch die Menge `Clauses` unlösbar sein.

Wir diskutieren nun die Hilfsprozeduren, die bei der Implementierung der Funktion `solve` verwendet wurden. Als erstes besprechen wir die Funktion `saturate`. Diese Funktion erhält eine Menge S von Klauseln als Eingabe und führt alle möglichen Unit-Schnitte und Unit-Subsumptionen durch. Die Funktion `saturate` ist in Abbildung 6.14 auf Seite 102 gezeigt.

```
1  def saturate(Clauses):
2      S     = Clauses.copy()
3      Units = { C for C in S if len(C) == 1 }
4      Used  = set()
5      while len(Units) > 0:
6          unit  = Units.pop()
7          Used |= { unit }
8          l     = arb(unit)
9          S     = reduce(S, l)
10         Units = { C for C in S if len(C) == 1 } - Used
11     return S
```

Abbildung 6.14: Die Funktion `saturate`

Die Implementierung von `saturate` funktioniert wie folgt:

1. Zunächst kopieren wir die Menge `Clauses` in die Variable `S`. Dies ist notwendig, da wir die Menge `S` später verändern werden. Die Funktion `saturate` soll das Argument `Clauses` aber nicht verändern.
2. Dann berechnen wir in Zeile 3 die Menge `Units` aller Unit-Klauseln.
3. Anschließend initialisieren wir in Zeile 4 die Menge `Used` als die leere Menge. In dieser Menge merken wir uns, welche Unit-Klauseln wir schon für Unit-Schnitte und Subsumptionen benutzt haben.
4. Solange die Menge `Units` der Unit-Klauseln nicht leer ist, wählen wir in Zeile 6 mit Hilfe der Funktion `pop` eine beliebige Unit-Klausel `unit` aus der Menge `Units` aus.
5. In Zeile 7 fügen wir die Klausel `unit` zu der Menge `Used` der benutzten Klausel hinzu.
6. In Zeile 8 extrahieren mit der Funktion `arb` das Literal `l` der Klausel `Unit`. Die Funktion `arb` liefert ein beliebiges Element der Menge zurück, das dieser Funktion als Argument übergeben wird. Enthält diese Menge nur ein Element, so wird also dieses Element zurück gegeben.
7. In Zeile 9 wird die eigentliche Arbeit durch einen Aufruf der Funktion `reduce` geleistet. Diese Funktion berechnet alle Unit-Schnitte, die mit der Unit-Klausel $\{\texttt{l}\}$ möglich sind und entfernt darüber hinaus alle Klauseln, die durch die Unit-Klausel $\{\texttt{l}\}$ subsumiert werden.
8. Wenn die Unit-Schnitte mit der Unit-Klausel $\{\texttt{l}\}$ berechnet werden, können neue Unit-Klauseln entstehen, die wir in Zeile 10 aufsammeln. Wir sammeln dort aber nur die Unit-Klauseln auf, die wir noch nicht benutzt haben.
9. Die Schleife in den Zeilen 5 – 10 wird nun solange durchlaufen, wie wir Unit-Klauseln finden, die wir noch nicht benutzt haben.

10. Am Ende geben wir die verbliebene Klauselmenge als Ergebnis zurück.

Die dabei verwendete Funktion `reduce` ist in Abbildung 6.15 gezeigt. Im vorigen Abschnitt hatten wir die Funktion *reduce*(S, l), die eine Klausel-Menge `Cs` mit Hilfe des Literals l reduziert, als

$$\mathit{reduce}(\texttt{Cs}, l) = \Big\{ C \backslash \{\overline{l}\} \mid C \in \texttt{Cs} \wedge \overline{l} \in C \Big\} \cup \Big\{ C \in \texttt{Cs} \mid \overline{l} \notin C \wedge l \notin C \Big\} \cup \Big\{ \{l\} \Big\}$$

definiert. Die Implementierung setzt diese Definition unmittelbar um.

```
1  def reduce(Clauses, l):
2      lBar = complement(l)
3      return   { C - { lBar } for C in Clauses if lBar in C }          \
4             | { C for C in Clauses if lBar not in C and l not in C }  \
5             | { frozenset({l}) }
```

Abbildung 6.15: Die Funktion `reduce`

Die Implementierung des Algorithmus von Davis und Putnam benutzt außer den bisher diskutierten Funktionen noch zwei weitere Hilfsprozeduren, deren Implementierung in Abbildung 6.16 auf Seite 103 gezeigt wird.

1. Die Funktion `selectLiteral` wählt eine beliebige Variable aus einer gegeben Menge `Clauses` von Klauseln aus, das außerdem nicht in der Menge `Forbidden` von den Variablen vorkommen darf, die bereits benutzt worden sind. Dazu iterieren wir zunächst über alle Klauseln $C \in$ `Clauses` und dann über alle Literale l der Klauseln C. Aus diesen Literalen extrahieren wir die darin enthaltene Variable mit Hilfe der Funktion `extractVariable`. Anschließend wird eine beliebige Variable zurück gegeben.
2. Die Funktion `arb` gibt ein nicht näher spezifiziertes Element einer Menge zurück.

```
1  def selectLiteral(Clauses, Forbidden):
2      Variables = { extractVariable(l) for C in Clauses for l in C } - Forbidden
3      return arb(Variables)
4
5  def arb(S):
6      "Return some member from the set S."
7      for x in S:
8          return x
```

Abbildung 6.16: Die Funktionen `select` und `negateLiteral`

Die oben dargestellte Version des Verfahrens von Davis und Putnam lässt sich in vielerlei Hinsicht verbessern. Aus Zeitgründen können wir auf solche Verbesserungen nicht weiter eingehen. Der interessierte Leser sei hier auf die folgende Arbeit von Moskewicz et.al. [MMZ+01] verwiesen:

> *Chaff: Engineering an Efficient SAT Solver*
> von M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik

6.7 Das 8-Damen-Problem

In diesem Abschnitt zeigen wir, wie bestimmte kombinatorische Problem als aussagenlogische Probleme formuliert werden können. Diese können dann anschließend mit dem Algorithmus von Davis und Putnam gelöst werden. Als konkretes Beispiel betrachten wir das 8-Damen-Problem. Dabei geht es darum, 8 Damen so auf

einem Schach-Brett aufzustellen, dass keine Dame eine andere Dame schlagen kann. Beim Schach-Spiel kann eine Dame dann eine andere Figur schlagen, wenn diese Figur entweder

- in derselben Zeile,
- in derselben Spalte oder
- in derselben Diagonale

wie die Dame steht. Abbildung 6.17 auf Seite 104 zeigt ein Schachbrett, in dem sich in der dritten Zeile in der vierten Spalte eine Dame befindet. Diese Dame kann auf alle die Felder ziehen, die mit Pfeilen markierte sind, und kann damit Figuren, die sich auf diesen Feldern befinden, schlagen.

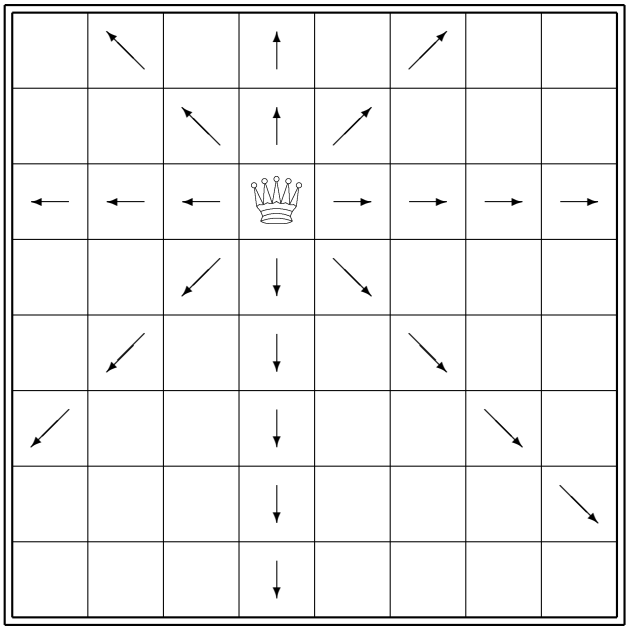

Abbildung 6.17: Das 8-Damen-Problem

Als erstes überlegen wir uns, wie wir ein Schach-Brett mit den darauf positionierten Damen aussagenlogisch repräsentieren können. Eine Möglichkeit besteht darin, für jedes Feld eine aussagenlogische Variable einzuführen. Diese Variable drückt aus, dass auf dem entsprechenden Feld eine Dame steht. Wir ordnen diesen Variablen wie folgt Namen zu: Die Variable, die das j-te Feld in der i-ten Zeile bezeichnet, stellen wir durch den String

$$\texttt{'Q(}i\texttt{,}j\texttt{)'} \quad \text{mit } i, j \in \{1, \cdots, 8\}$$

dar. Wir nummerieren die Zeilen dabei von oben beginnend von 1 bis 8 durch, während die Spalten von links nach rechts numeriert werden. Abbildung 6.19 auf Seite 105 zeigt die Zuordnung der Variablen zu den Feldern. Die in Abbildung 6.18 gezeigte Funktion $\texttt{var}(r, c)$ berechnet die Variable, die ausdrückt, dass sich in Zeile r und Spalte c eine Dame befindet.

```
1  def var(row, col):
2      return 'Q' + str(row) + ',' + str(col) + ')'
```

Abbildung 6.18: Die Funktion `var` Zur Berechnung der aussagenlogischen Variablen

Als nächstes überlegen wir uns, wie wir die einzelnen Bedingungen des 8-Damen-Problems als aussagenlogische Formeln kodieren können. Letztlich lassen sich alle Aussagen der Form

Q(1,1)	Q(1,2)	Q(1,3)	Q(1,4)	Q(1,5)	Q(1,6)	Q(1,7)	Q(1,8)
Q(2,1)	Q(2,2)	Q(2,3)	Q(2,4)	Q(2,5)	Q(2,6)	Q(2,7)	Q(2,8)
Q(3,1)	Q(3,2)	Q(3,3)	Q(3,4)	Q(3,5)	Q(3,6)	Q(3,7)	Q(3,8)
Q(4,1)	Q(4,2)	Q(4,3)	Q(4,4)	Q(4,5)	Q(4,6)	Q(4,7)	Q(4,8)
Q(5,1)	Q(5,2)	Q(5,3)	Q(5,4)	Q(5,5)	Q(5,6)	Q(5,7)	Q(5,8)
Q(6,1)	Q(6,2)	Q(6,3)	Q(6,4)	Q(6,5)	Q(6,6)	Q(6,7)	Q(6,8)
Q(7,1)	Q(7,2)	Q(7,3)	Q(7,4)	Q(7,5)	Q(7,6)	Q(7,7)	Q(7,8)
Q(8,1)	Q(8,2)	Q(8,3)	Q(8,4)	Q(8,5)	Q(8,6)	Q(8,7)	Q(8,8)

Abbildung 6.19: Zuordnung der Variablen

- “in einer Zeile steht höchstens eine Dame”,
- “in einer Spalte steht höchstens eine Dame”, oder
- “in einer Diagonale steht höchstens eine Dame”

auf dasselbe Grundmuster zurückführen: Ist eine Menge von aussagenlogischen Variablen

$$V = \{x_1, \cdots, x_n\}$$

gegeben, so brauchen wir eine Formel die aussagt, dass höchstens eine der Variablen aus V den Wert `True` hat. Das ist aber gleichbedeutend damit, dass für jedes Paar $x_i, x_j \in V$ mit $x_i \neq x_j$ die folgende Formel gilt:

$$\neg(x_i \wedge x_j).$$

Diese Formel drückt aus, dass die Variablen x_i und x_j nicht gleichzeitig den Wert `True` annehmen. Nach den

DeMorgan'schen Gesetzen gilt

$$\neg(x_i \wedge x_j) \leftrightarrow \neg x_i \vee \neg x_j$$

und die Klausel auf der rechten Seite dieser Äquivalenz schreibt sich in Mengen-Schreibweise als

$$\{\neg x_i, \neg x_j\}.$$

Die Formel, die für eine Variablen-Menge V ausdrückt, dass keine zwei verschiedenen Variablen gleichzeitig wahr sind, kann daher als Klausel-Menge in der Form

$$\big\{\, \{\neg p, \neg q\} \mid p \in V \wedge\ q \in V \wedge p \neq q \big\}$$

geschrieben werden. Wir setzen diese Überlegungen in eine *Python*-Funktion um. Die in Abbildung 6.20 gezeigte Funktion `atMostOne()` bekommt als Eingabe eine Menge S von aussagenlogischen Variablen. Der Aufruf `atMostOne`(S) berechnet eine Menge von Klauseln. Diese Klauseln sind genau dann wahr, wenn höchstens eine der Variablen aus S den Wert `True` hat.

```
1  def atMostOne(S):
2      return { frozenset({('¬',p), ('¬', q)}) for p in S
3                                              for q in S
4                                              if  p != q
5             }
```

Abbildung 6.20: Die Funktion `atMostOne`

Mit Hilfe der Funktion `atMostOne` können wir nun die Funktion `atMostOneInRow` implementieren. Der Aufruf

```
atMostOneInRow(row, n)
```

berechnet für eine gegebene Zeile `row` bei einer Brettgröße von `n` eine Formel, die ausdrückt, dass in der Zeile `row` höchstens eine Dame steht. Abbildung 6.21 zeigt die Funktion `atMostOneInRow`: Wir sammeln alle Variablen der durch `row` spezifizierten Zeile in der Menge

$$\big\{\texttt{var}(\texttt{row}, j) \mid j \in \{1, \cdots, n\}\big\}$$

auf und rufen mit dieser Menge die Funktion `atMostOne`$()$ auf, die das Ergebnis als Menge von Klauseln liefert.

```
1  def atMostOneInRow(row, n):
2      return atMostOne({ var(row, col) for col in range(1,n+1) })
```

Abbildung 6.21: Die Funktion `atMostOneInRow`

Als nächstes berechnen wir eine Formel die aussagt, dass mindestens eine Dame in einer gegebenen Spalte steht. Für die erste Spalte hätte diese Formel die Form

$$\texttt{Q}(1,1) \vee \texttt{Q}(2,1) \vee \texttt{Q}(3,1) \vee \texttt{Q}(4,1) \vee \texttt{Q}(5,1) \vee \texttt{Q}(6,1) \vee \texttt{Q}(7,1) \vee \texttt{Q}(8,1)$$

und wenn allgemein eine Spalte c mit $c \in \{1, \cdots, 8\}$ gegeben ist, lautet die Formel

$$\texttt{Q}(1,c) \vee \texttt{Q}(2,c) \vee \texttt{Q}(3,c) \vee \texttt{Q}(4,c) \vee \texttt{Q}(5,c) \vee \texttt{Q}(6,c) \vee \texttt{Q}(7,c) \vee \texttt{Q}(8,c).$$

Schreiben wir diese Formel in der Mengenschreibweise als Menge von Klauseln, so erhalten wir

$$\big\{\{\texttt{Q}(1,c), \texttt{Q}(2,c), \texttt{Q}(3,c), \texttt{Q}(4,c), \texttt{Q}(5,c), \texttt{Q}(6,c), \texttt{Q}(7,c), \texttt{Q}(8,c)\}\big\}.$$

Abbildung 6.22 zeigt eine *Python*-Funktion, die für eine gegebene Spalte `col` und eine gegebene Brettgröße `n` die entsprechende Klausel-Menge berechnet. Der Schritt, von einer einzelnen Klausel zu einer Menge von Klauseln überzugehen ist notwendig, denn unsere Implementierung des Algorithmus von Davis und Putnam

arbeitet mit einer Menge von Klauseln.

```
1  def oneInColumn(col, n):
2      return { frozenset({ var(row, col) for row in range(1,n+1) }) }
```

Abbildung 6.22: Die Funktion `oneInColumn`

An dieser Stelle erwarten Sie vielleicht, dass wir noch Formeln angeben die ausdrücken, dass in einer gegebenen Spalte höchstens eine Dame steht und dass in jeder Zeile mindestens eine Dame steht. Solche Formeln sind aber unnötig, denn wenn wir wissen, dass in jeder Spalte mindestens eine Dame steht, so wissen wir bereits, dass auf dem Brett mindestens 8 Damen stehen. Wenn wir nun zusätzlich wissen, dass in jeder Zeile höchstens eine Dame steht, so ist automatisch klar, dass höchstens 8 Damen auf dem Brett stehen. Damit stehen also insgesamt genau 8 Damen auf dem Brett. Dann kann aber in jeder Spalte nur höchstens eine Dame stehen, denn sonst hätten wir mehr als 8 Damen auf dem Brett und genauso muss in jeder Zeile mindestens eine Dame stehen, denn sonst würden wir in der Summe nicht auf 8 Damen kommen.

Als nächstes überlegen wir uns, wie wir die Variablen, die auf derselben Diagonale stehen, charakterisieren können. Es gibt grundsätzlich zwei verschiedene Arten von Diagonalen: Absteigende Diagonalen und aufsteigende Diagonalen. Wir betrachten zunächst die aufsteigenden Diagonalen. Die längste aufsteigende Diagonale, wir sagen dazu auch Hauptdiagonale, besteht im Fall eines 8×8-Bretts aus den Variablen

$$\mathtt{Q}(8,1),\ \mathtt{Q}(7,2),\ \mathtt{Q}(6,3),\ \mathtt{Q}(5,4),\ \mathtt{Q}(4,5),\ \mathtt{Q}(3,6),\ \mathtt{Q}(2,7),\ \mathtt{Q}(1,8).$$

Die Indizes r und c der Variablen $\mathtt{Q}(r,c)$ erfüllen offenbar die Gleichung

$$r + c = 9.$$

Allgemein erfüllen die Indizes der Variablen einer aufsteigenden Diagonale, die mehr als ein Feld enthält, die Gleichung

$$r + c = k,$$

wobei k im Falle eines 8×8 Schach-Bretts einen Wert aus der Menge $\{3, \cdots, 15\}$ annimmt. Den Wert k geben wir als Argument bei der Funktion `atMostOneInRisingDiagonal` mit. Diese Funktion ist in Abbildung 6.23 gezeigt.

```
1  def atMostOneInRisingDiagonal(k, n):
2      S = { var(row, col) for row in range(1, n+1)
3                          for col in range(1, n+1)
4                          if  row + col == k
5          }
6      return atMostOne(S)
```

Abbildung 6.23: Die Funktion `atMostOneInUpperDiagonal`

Um zu sehen, wie die Variablen einer fallenden Diagonale charakterisiert werden können, betrachten wir die fallende Hauptdiagonale, die aus den Variablen

$$\mathtt{Q}(1,1),\ \mathtt{Q}(2,2),\ \mathtt{Q}(3,3),\ \mathtt{Q}(4,4),\ \mathtt{Q}(5,5),\ \mathtt{Q}(6,6),\ \mathtt{Q}(7,7),\ \mathtt{Q}(8,8)$$

besteht. Die Indizes r und c dieser Variablen erfüllen offenbar die Gleichung

$$r - c = 0.$$

Allgemein erfüllen die Indizes der Variablen einer absteigenden Diagonale die Gleichung

$$r - c = k,$$

wobei k einen Wert aus der Menge $\{-6, \cdots, 6\}$ annimmt. Den Wert k geben wir als Argument bei der Funktion

`atMostOneInLowerDiagonal` mit. Diese Funktion ist in Abbildung 6.24 gezeigt.

```
1  def atMostOneInFallingDiagonal(k, n):
2      S = { var(row, col) for row in range(1, n+1)
3                          for col in range(1, n+1)
4                          if  row - col == k
5          }
6      return atMostOne(S)
```

Abbildung 6.24: Die Funktion `atMostOneInLowerDiagonal`

Jetzt sind wir in der Lage, unsere Ergebnisse zusammen zu fassen: Wir können eine Menge von Klauseln konstruieren, die das 8-Damen-Problem vollständig beschreiben. Abbildung 6.25 zeigt die Implementierung der Funktion `allClauses`. Der Aufruf

`allClauses`(n)

rechnet für ein Schach-Brett der Größe n eine Menge von Klauseln aus, die genau dann erfüllt sind, wenn auf dem Schach-Brett

1. in jeder Zeile höchstens eine Dame steht (Zeile 2),
2. in jeder absteigenden Diagonale höchstens eine Dame steht (Zeile 3),
3. in jeder aufsteigenden Diagonale höchstens eine Dame steht (Zeile 4) und
4. in jeder Spalte mindestens eine Dame steht (Zeile 5).

Die Ausdrücke in den einzelnen Zeilen liefern Listen, deren Elemente Klausel-Mengen sind. Was wir als Ergebnis brauchen, ist aber eine Klausel-Menge und keine Liste von Klausel-Mengen. Daher wandeln wir in Zeile 6 die Liste `All` in eine Menge von Klauseln um.

```
1  def allClauses(n):
2      All = [ atMostOneInRow(row, n)           for row in range(1, n+1)        ] \
3          + [ atMostOneInFallingDiagonal(k, n) for k in range(3, (2*n-1)+1)    ] \
4          + [ atMostOneInRisingDiagonal(k, n)  for k in range(-(n-2), (n-2)+1) ] \
5          + [ oneInColumn(col, n)              for col in range(1, n+1)        ]
6      return { clause for S in All for clause in S }
```

Abbildung 6.25: Die Funktion `allClauses`

Als letztes zeigen wir in Abbildung 6.26 die Funktion `queens`, mit der wir das 8-Damen-Problem lösen können.

1. Zunächst kodieren wir das Problem als eine Menge von Klauseln, die genau dann lösbar ist, wenn das Problem eine Lösung hat.
2. Anschließend berechnen wir die Lösung mit Hilfe der Funktion `sovle` aus dem Modul `davisPutnam`, das wir als `dp` importiert haben.
3. Zum Schluss wird die berechnete Lösung mit Hilfe der Funktion `printBoard` ausgedruckt.

   Hierbei ist `printBoard` eine Funktion, welche die Lösung in lesbarere Form ausdruckt. Das funktioniert allerdings nur, wenn ein Font verwendet wird, bei dem alle Zeichen die selbe Breite haben. Diese Funktion ist der Vollständigkeit halber in Abbildung 6.27 gezeigt, wir wollen die Implementierung aber nicht weiter diskutieren.

Das vollständige Programm finden Sie als Jupyter Notebook auf meiner Webseite unter dem Namen N-Queens.ipynb.

```
def queens(n):
    "Solve the n queens problem."
    Clauses  = allClauses(n)
    Solution = dp.solve(Clauses, set())
    if Solution != { frozenset() }:
        printBoard(Solution, n)
    else:
        print(f'The problem is not solvable for {n} queens!')
```

Abbildung 6.26: Die Funktion queens

```
def printBoard(I, n):
    if I == { frozenset() }:
        return
    print("-" * (8*n+1))
    for row in range(1, n+1):
        printEmptyLine(n)
        line = "|";
        for col in range(1, n+1):
            if frozenset({ var(row, col) }) in I:
                line += "   Q   |"
            else:
                line += "       |"
        print(line)
        printEmptyLine(n)
        print("-" * (8*n+1))

def printEmptyLine(n):
    line = "|"
    for col in range(1, n+1):
        line += "       |"
    print(line)
```

Abbildung 6.27: Die Funktion printBoard()

Die durch den Aufruf solve(*Clauses*, {}) berechnete Menge solution enthält für jede der Variablen 'Q(*r*,*c*)' entweder die Unit-Klausel {'Q(*r*,*c*)'} (falls auf diesem Feld eine Dame steht) oder aber die Unit-Klausel {('¬', 'Q(*r*,*c*)')} (falls das Feld leer bleibt). Eine graphische Darstellung einer berechneten Lösungen sehen Sie in Abbildung 6.28.

Das 8-Damen-Problem ist natürlich nur eine spielerische Anwendung der Aussagen-Logik. Trotzdem zeigt es die Leistungsfähigkeit des Algorithmus von Davis und Putnam sehr gut, denn die Menge der Klauseln, die von der Funktion allClauses berechnet wird, besteht aus 512 verschiedenen Klauseln. In dieser Klausel-Menge kommen 64 verschiedene Variablen vor.

In der Praxis gibt es viele Probleme, die sich in ganz ähnlicher Weise auf die Lösung einer Menge von Klauseln zurückführen lassen. Dazu gehört zum Beispiel das Problem, einen Stundenplan zu erstellen, der gewissen Nebenbedingungen genügt. Verallgemeinerungen des Stundenplan-Problems werden in der Literatur als Scheduling-Probleme bezeichnet. Die effiziente Lösung solcher Probleme ist Gegenstand der aktuellen Forschung.

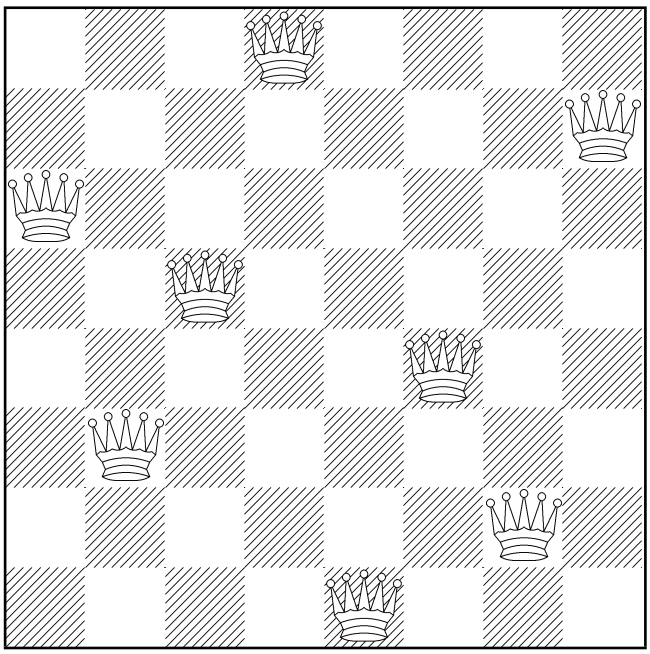

Abbildung 6.28: Eine Lösung des 8-Damen-Problems

6.8 Reflexion

1. Wie haben wir die Menge der aussagenlogischen Formeln definiert?
2. Wie ist die Semantik der aussagenlogischen Formeln festgelegt worden?
3. Wie können wir aussagenlogische Formeln in *Python* darstellen?
4. Was ist eine Tautologie?
5. Was ist eine konjunktive Normalform?
6. Wie können Sie die konjunktive Normalform einer gegebenen aussagenlogischen Formel berechnen und wie lässt sich diese Berechnung in *Python* implementieren?
7. Wie haben wir den Beweis-Begriff $M \vdash C$ definiert?
8. Welche Eigenschaften hat der Beweis-Begriff $\vdash$?
9. Wann ist eine Menge von Klauseln lösbar?
10. Wie funktioniert das Verfahren von Davis und Putnam?

Literaturverzeichnis

[Can95] Georg Cantor. Beiträge zur Begründung der transfiniten Mengenlehre. *Mathematische Annalen*, 46:481–512, 1895.

[Ced18] Naomi R. Ceder. *The Quick Python Book*. Manning Publications, 3rd edition, 2018.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[Lip98] Seymour Lipschutz. *Set Theory and Related Topics*. McGraw-Hill, New York, 1998.

[Lut13] Mark Lutz. *Learning Python*. O'Reilly and Associates, 5th edition, 2013.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 83, 1953.

[Tur36] Alan M. Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.