Theoretical Computer Science I: An Introduction to Logic via *Python*

— Winter 2019 —

Baden-Wuerttemberg Cooperative State University (DHBW)

Prof. Dr. Karl Stroetmann

October 11, 2019

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

`https://github.com/karlstroetmann/Logic`.

The lecture notes can be found in the directory `Lecture-Notes` in the file `logic.pdf`. The Jupyter Notebooks discussed in this lecture are found in the directory `Python`. As my knowledge of *Python* is constantly improving, these lecture notes are being revised regularly. To automatically update the lecture notes, you can install the program `git`. Then, using the command line of your favourite operating system, you can *clone* my repository using the command

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

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

```
git pull.
```

Contents

Chapter 1

Introduction

In this short chapter, I would like to motivate the reason that you have to learn mathematical logic to become a good computer scientist. 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. Today it is quite common that complex software projects require more than a thousand collaborating developers to develop a new system. The failure of a project of this size is very costly. The page

Staggering Impact of IT Systems Gone Wrong

presents several examples showing big software projects that have failed and have subsequently caused huge financial losses. To present just one recent example, the consolidation of Germany's federal IT system is currently in a crisis: Whereas the costs had originally been estimated at 1 billion euros, the current estimate is at 3.42 billion euros. This and numerous other 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. Then your ability to quickly grasp new concepts will be much more important 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 sets together with the most important operations defined on sets. Therefore it is suitable to implement most of the abstract ideas presented in this lecture. According to the IEEE (Institute of Electrical and Electronics Egineers), *Python* is the most popular programming language. Furthermore, *Python* is 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

   "$\neg$", "$\wedge$", "$\vee$", "$\rightarrow$" und "$\leftrightarrow$",

   while *first-order logic* also investigates the quantifiers

   "$\forall$" and "$\exists$",

   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.

Chapter 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* natural numbers via the property

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

Using this property, the set of even natural 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. We define the property $p(x)$ as

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

and proceed to 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

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

So we have shown that

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

holds. In other words, the set R is a member of itself if and only if R is not a member of itself! 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

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

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 less powerful than the axiom of comprehension, but, nevertheless, these methods will be sufficient for our purposes. We will continue to use the notation underlying the comprehension axiom and write set definitions in the form

$M = \{x \mid p(x)\}.$ (Read: M is the set of all x such that $p(x)$ holds.)

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 characters. 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 dots notation to define a set. Using dots 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 dots 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 which 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. This is summarized as the statement that

> a set abstracts from the order of its 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 some sets that have an infinite number of elements. One example is the set of natural numbers, which is 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.

This is the inductive definition of the set of natural numbers. We write

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

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

[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 its content is not freely available, at least not legally.

1. $\mathbb{N}^*$ is the set of positive natural numbers, 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.

   This set comprises all numbers that you know from school. For example, besides the rational numbers, this set contains the numbers $\sqrt{2}$ or π. 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 (Deutsch: Aussonderungs-Axiom), 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 $p(x)$:

$$N := \{x \in M \mid p(x)\} \qquad \text{(Read: } N \text{ is the set of all } x \text{ from } M \text{ that satisfy } p(x).\text{)}$$

Therefore, the axiom of specification states that if M is a set and $p(x)$ is a property that is either true or false for elements x of M, then

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

is a set. 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 natural numbers can be defined as

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

Exercise 1: Define the set S of all square numbers, i.e. the set S should contain the numbers 0, 1, 4, 9, and so on. $\diamond$

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

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

Example: We have

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

Furthermore, for any set M we have that

$$\emptyset \subseteq M.$$ $\diamond$

The power set of a set M is defined as the set of all subsets of M. This set is written as 2^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

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

This set has $8 = 2^3$ elements. $\diamond$

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.

Exercise 2: Compute the set $2^{\{1,2,3,4\}}$. $\diamond$

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

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

Note that $M \cup N$ is written as $\{1,2,3,5\}$ and not as $\{1,2,2,3,3,5\}$. An object x is either an element of a set or it isn't and hence it does not make sense to list an element more than once. $\diamond$

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 3: 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 4: Assume that M is a subset of $\mathbb{N}$. Compute the set $\bigcap 2^M$. ◇

Exercise 5: The sets A and B are defined as follows.

$$A := \{x \in \mathbb{N} \mid x \,\%\, 2 = 0\} \quad \text{and} \quad B := \{x \in \mathbb{N} \mid x \,\%\, 3 = 0\}.$$

(For two natural numbers x and k, the notation $x \,\%\, k$ is ued in computer science to denote the remainder that is left when x is divided by k.) How can we write the set $A \cap B$ using the axiom of specification? ◇

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 using 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

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

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:

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

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

Exercise 6: Assume that M and N are finite sets. How can the expression $card(M \times N)$ be reduced to an expression containing the expressions $card(M)$ and $card(N)$? ◇

2.10 Equality of Sets

We have now presented all the methods that we will use in this lecture 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

You should be able to answer the following questions without consulting the text.

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.

[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.

5. What is the axiom of extensionality?

If you want to develop a deeper understand of set theory, I recommend the book

Set Theory and Related Topics

by Seymour Lipschutz [Lip98].

Chapter 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 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*. If you have instead installed version 3.7 the programs should also work.

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.
>>>
```

Figure 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]

[1] While I am usually in the habit of terminating every sentence with either a full stop, a question mark, or an exclamation

```
python
```

After the interpreter is started, the user sees the output that is shown in Figure 3.1 on page 16. The string ">>>" is the prompt. It signals that the interpreter is waiting for input. If we input the string

```
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 that we do not need a command line to execute *Python* scripts, since we even can run *Python* in a browser in so called jupyter notebooks. If you have installed *Python* by means of the Anaconda distribution, then you already have installed these notebooks. 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 off 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
```

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.

```
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
000000000000000000000000000000000000000000000000
```

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 = "

followed 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 $\sum\limits_{i=1}^{n} i$.

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. In this case, the function has the name sum. 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 and indentation matters.

   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 $\sum_{i=1}^{n} i$ for any natural number n 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` or the operator "`|`":

```
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` or the operator "`&`":

```
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

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

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[2] 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}
```

[2]In mathematics it is common to write "iff" as an abbreviation for "if and only if"

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 immutable. 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 an 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@www 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)
```

If we want to create a tuple of length 1, we have to use the following syntax:

```
L = (x,)
```

Note that in the expression above the comma is not optional as the expression (x) would be interpreted as x.

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
```

The functions "`min`" and "`max`" also work for sets.

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
```

Next, we show the table for the operator "`or`". 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 <= 7:
             print(x)
             x += 1
```

```
1
2
3
4
5
6
7
```

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 multiples 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 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/3.7/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:

   $$\texttt{sort([])} = \texttt{[]}.$$

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:

   $$\texttt{sort(L)} = \texttt{[min(L)] + sort([}x \in \texttt{L} \,|\, x \neq \texttt{min}(L)\texttt{])}.$$

Figure 3.2 on page 36 shows the program `min-sort.py` that implements selection sort in *Python*.

```
def minSort(L):
    if L == []:
        return []
    m = min(L)
    return [m] + minSort([x for x in L if x != m])

L = [ 2, 13, 5, 13, 7, 2, 4 ]
print('minSort(', L, ') = ', minSort(L), sep='')
```

Figure 3.2: Implementing selection sort in *Python*.

3.3 Loading a Program

The *Python* 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 36. 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
5945713821785251664274
```

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:

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

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

$$f_R : A \to B$$

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/3.7/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 *Python* data types have been introduced in this chapter?

2. What are the different ways to define a set in *Python*?

3. How can you build lists and sets via iterators?
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. What type of control structures are supported in *Python*?

Chapter 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 45 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 $(x_n)_{n\in\mathbb{N}}$ inductively as follows:

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

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 46 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}

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

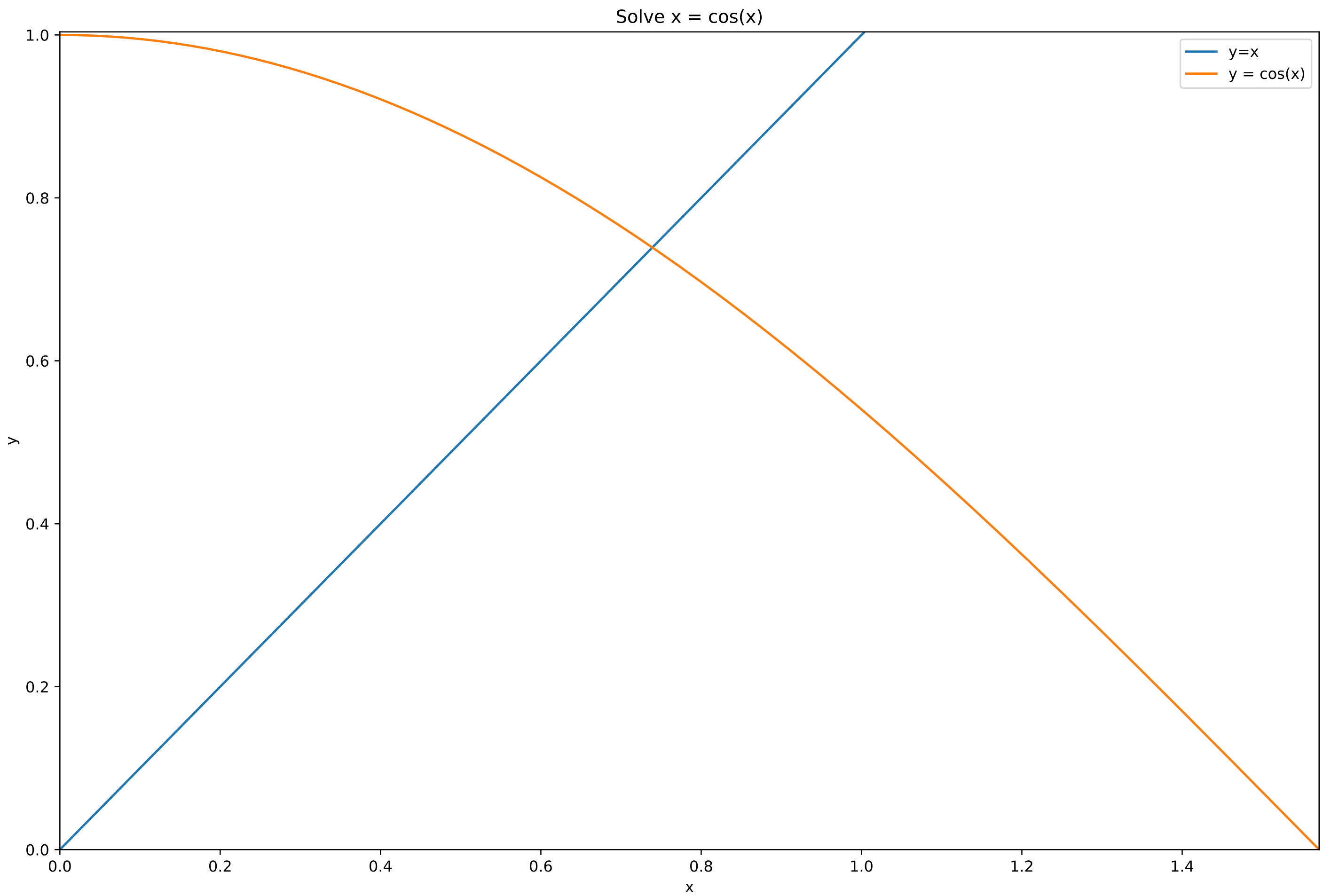

Figure 4.1: The functions $y = x$ and $y = cos(x)$.

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 46 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

   $$f(x) = x.$$

   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

```
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
```

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

```
from math import cos

def solve(f, x0):
    """
    Solve the equation f(x) = x using a fixed point iteration.
    x0 is the start value.
    """
    x = x0
    for n in range(10000):  # at most 10000 iterations
        oldX = x;
        x    = f(x);
        if abs(x - oldX) < 1.0e-15:
            return x;

print("solution to x = cos(x): ", solve(cos, 0));
print("solution to x = 1/(1+x):", solve(lambda x: 1/(1+x), 0));
```

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

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

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. $\diamond$

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

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

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

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

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

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

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))
```

Figure 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 ♣ (club),
   (b) "h" is short for ♡ (hearts),
   (c) "d" is short for ♢ (diamonds), and
   (d) "s" is short for ♠ (spades).

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 $\langle v, s\rangle$. 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 *directed 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 49. 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.

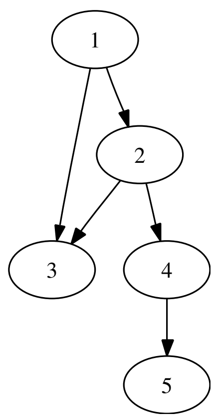

Figure 4.5: A simple graph.

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

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

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 $n-1$, since there are $n-1$ direct connections of the form $\langle x_i, x_{i+1} \rangle$ that make up this path. To put it differently, 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 on the path.

Furthermore, two vertices a and b of a graph 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. We have previously defined a binary relation 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

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

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: $n \mapsto n + 1$

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

The idea behind this definition is that if $\langle x, z \rangle \in R^n$, then there is a path of length n that starts in x and ends in z.

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

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

   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

   $$\underbrace{\underbrace{x \mapsto \cdots \mapsto y}_{\text{path of length } n.} \mapsto z.}_{\text{path of length } n+1.}$$

   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 $R \cup (R^+ \circ R)$. In order to prove equation 4.1 we first note that the following law of distributivity holds:

$$(P \cup Q) \circ R = (P \circ R) \cup (Q \circ R).$$

(A proof of this equation can be found in my lecture notes on linear algebra on page 42.) Using this law, we can prove equation 4.1 algebraically. We have:

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

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 $n \in \mathbb{N}$. As $R^1 = R$, the base case $n = 0$ of this proof is immediate from the definition of T_0. In the induction step we observe the following:

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

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:

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

If the relation R is finite, then the transitive closure R^+ is finite, too. All of the sets T_n are 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 54 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 `product`(R_1, R_2) computes the relational product $R_1 \circ R_2$ according to the formula

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

```
def product(R1, R2):
    'Compute the relational product of R1 and R2.'
    return { (x, z) for (x, y1) in R1 for (y2, z) in R2 if y1 == y2 }

def transClosure(R):
    'Compute the transitive closure of the binary relation R.'
    T = R
    while True:
        oldT = T
        T    = product(R,T) | R
        if T == oldT:
            return T

R = { (1,2), (2,3), (1,3), (2,4), (4,5) }
print( 'R  = ', R );
print( 'Computing the transitive closure of R:' );
T = transClosure(R);
print( 'R+ = ', T );
```

Figure 4.6: Computing the transitive closure.

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:
$$\texttt{first}\bigl(\langle x_1, \cdots, x_m\rangle\bigr) = x_1.$$
2. Given a tuple T, the function $\texttt{last}(T)$ returns the last element of T:
$$\texttt{last}\bigl(\langle x_1, \cdots, x_m\rangle\bigr) = x_m.$$
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:
$$\mathcal{P}_1 \bullet \mathcal{P}_2 = \bigl\{\, T_1 \oplus T_2 \mid T_1 \in \mathcal{P}_1 \wedge T_2 \in \mathcal{P}_2 \wedge \texttt{last}(T_1) = \texttt{first}(T_2) \,\bigr\}.$$

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 55. 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 55 shows a cyclic graph. This graph is cyclic

```
def findPaths(R):
    P = R;
    while True:
        oldP = P
        P    = R | 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)
```

Figure 4.7: Computing all connections.

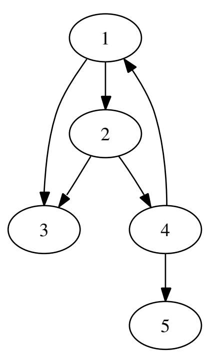

Figure 4.8: A graph with a cycle.

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 def pathProduct(P, Q):
2     return { join(S, T) for S in P for T in Q
3                         if S[-1] == T[0] and noCycle(S, T)
4            }
5
6 def noCycle(T1, T2):
7     return len({ x for x in T1 } & { x for x in T2 }) == 1
```

Figure 4.9: Computing the connections in a cyclic graph.

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 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 57 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 vertex `start` that have a length of at most 1, the second iteration computes all paths starting in vertex `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-th iteration unless a shorter path has already been found in a previous iteration.

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

```
def reachable(start, goal, R):
    P = { (start,) }
    while True:
        oldP  = P
        P     = P | 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) & set(T2)) == 1

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

Figure 4.10: Finding the shortest path between two vertices.

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'}.
```

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 states that are allowed according to the specification of the problem 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, where a problem is any situation where either the goat eats the cabbage or the wolf eats the goat. 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.

Formally, 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
```

Note that we have to use a line continuation backslash "\" at the end of the first line of the return statement. We do not need a continuation backslash at the end of the second line of the return statement since the opening parenthesis at the beginning of the second line has not yet been closed when the second line finishes and therefore *Python* is able to figure out that the expression defined in this line is continued in the third line.

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 `States`.
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 `B` is an element of the power set of `S`.
3. Therefore `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 the set `S-B` is an element of the set `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 | 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 4.11 on page 60 displays the relation R graphically. Figure 4.12 on page 60 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.

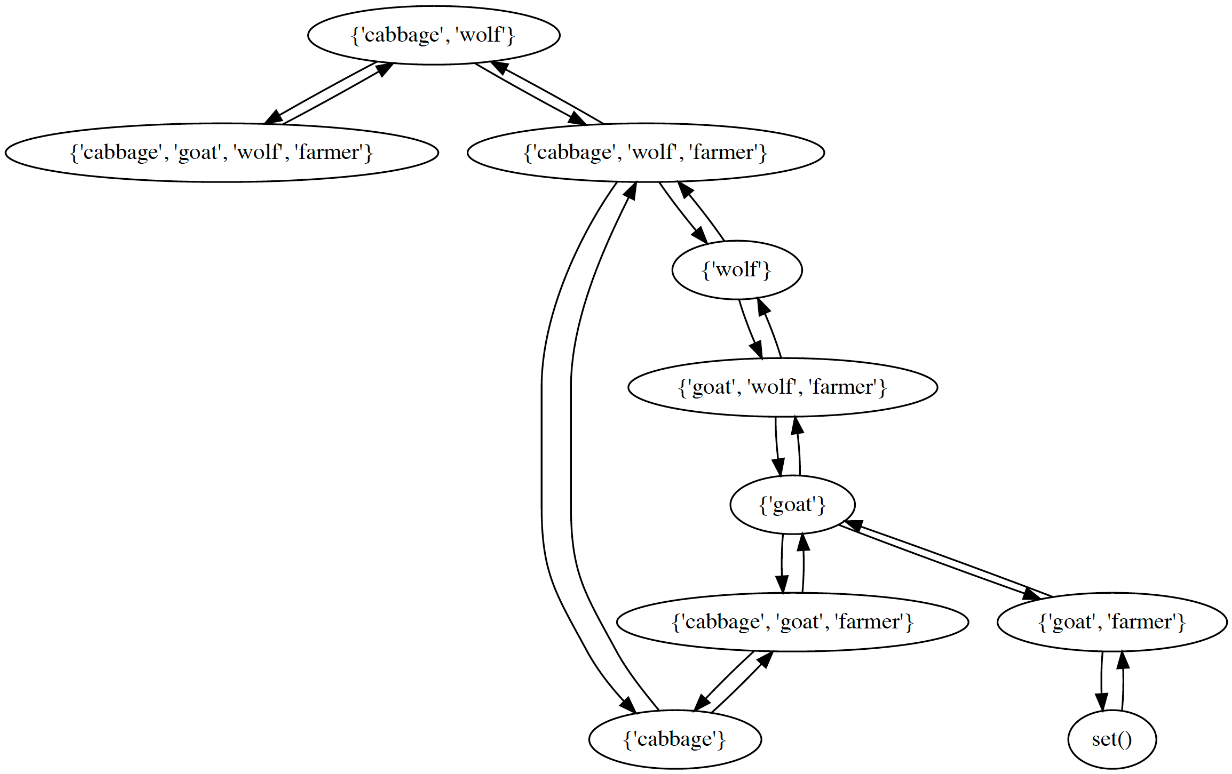

Figure 4.11: The relation R shown as a directed graph.

```
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 | R2
start = All
goal  = frozenset()
Path  = findPath(start, goal, R)
```

Figure 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'}
```

Figure 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 the string

`"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

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

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 in the second semester..

The function `diff` that is shown in Figure 4.14 on page 64 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
```

Figure 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)}')
```

Figure 4.15: Testing symbolic differentiation.

Chapter 5

Limits of Computability

Every discipline of the sciences has its limits: Students of the medical sciences soon realize that it is difficult to raise the dead and even religious zealots have trouble to walk on water. Similarly, computer science has its limits. We will discuss these limits next. First, we show that we cannot decide whether a computer program will eventually terminate or whether it will run forever. Second, we prove that it is impossible to atomatically check whether two functions are equivalent.

5.1 The Halting Problem

In this subsection we prove that it is not possible for a computer program to decide whether another computer program does terminate. This problem is known as the *halting problem*. Before we give a formal proof that the halting problem is undecidable, let us discuss one example that shows why it is indeed difficult to decide whether a program does always terminate. Consider the program shown in Figure 5.1 on page 66. This program contains an obvious infinite loop in line 6. However, this loop is only entered if the expression

$$\texttt{legendre}(n)$$

in line 5 evaluates to `false`. Given a natural number `n`, the expression $\texttt{legendre}(n)$ tests whether there is a prime between $\texttt{n}^2$ and $(\texttt{n}+1)^2$. If, however, the intervall

$$[n^2,(n+1)^2] := \{k \in \mathbb{N} \mid n^2 \leq k \wedge k \leq (n+1)^2\}$$

does not contain a prime number, then $\texttt{legendre}(n)$ evaluates to `false` for this value of n. The function `legendre` is defined in line 2. Given a natural number n, it returns `true` if and only if the formula

$$\exists k \in \mathbb{N} : n^2 \leq k \wedge k \leq (n+1)^2 \wedge \mathit{isPrime}(k)$$

holds true. The French mathematican Adrien-Marie Legendre (1752 – 1833) conjectured that for any natural number $n \in \mathbb{N}$ there is prime number p such that

$$n^2 \leq p \wedge p \leq (n+1)^2$$

holds. Although there are a number of reasons in support of Legendre's conjecture, to this day nobody has been able to prove it. The question, whether the invocation of the function f will terminate for every user input is, therefore, unknown as it depends on the truth of Legendre's conjecture: If we had some procedure that could check whether the function $\texttt{find\_counter\_example}(1)$ does terminate, then this procedure would be able to decide whether Legendre's theorem is true. Therefore, it should come as no surprise that such a procedure does not exist.

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

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

def legendre(n):
    k = n * n + 1;
    while k < (n + 1) ** 2:
        if is_prime(k):
            print(f'{n}**2 < {k} < {n+1}**2')
            return True
        k += 1
    return False

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
```

Figure 5.1: A program checking Legendre's conjecture.

Let us proceed to prove formally that the halting problem is not solvable. To this end, we need the following definition.

Definition 1 (Test Function) *A string t is a* test function with name f *iff t has the form*

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

and, furthermore, the string t can be parsed as a Python function, that is the evaluation of the expression

$\texttt{exec}(t)$

does not yield an error. The set of all test functions is denoted as TF. If $t \in$ TF and t has the name f, then this is written as

$\texttt{name}(t) = f.$ □

Examples:

1. We define the string s_1 as follows:

```
'''def simple(x):
       return 0
'''
```

Then s_1 is a test function with the name `simple`.

2. We define the string s_2 as

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

Then s_2 is a test function with the name `loop`.

3. We define the string s_3 as

```
'''def hugo(x):
       return ++x
'''
```

Then s_3 is not a test function. The reason is that *Python* does not support the operator "++". Therefore,

```
exec(s3)
```

yields an error message complaining about the two "+" characters.

In order to be able to formalize the halting problem succinctly, we introduce three additional notations.

Notation 2 ($\leadsto$, $\downarrow$, $\uparrow$) *If n is the name of a Python function that takes k arguments $a_1, \cdots, a_k$, then we write*

$$n(a_1, \cdots, a_k) \leadsto r$$

iff the evaluation of the expression $n(a_1, \cdots, a_k)$ yields the result r. If we are not concerned with the result r but only want to state that the evaluation terminates, then we will write

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

*and read this notation as "*evaluating $n(a_1, \cdots, a_k)$ terminates*". If the evaluation of the expression $n(a_1, \cdots, a_k)$ does <u>not</u> terminate, this is written as*

$$n(a_1, \cdots, a_k) \uparrow.$$

*This notation is read as "*evaluation of $n(a_1, \cdots, a_k)$ diverges*".* □

Beispiele: Using the test functions defined earlier, we have:

1. `simple("emil")` $\leadsto 0$,
2. `simple("emil")` $\downarrow$,
3. `loop(2)` $\uparrow$.

The *halting problem* for *Python* functions is the question whether there is a *Python* function

```
def stops(t, a):
    ⋮
```

that takes as input a test function t and a string a and that satisfies the following specification:

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

   If the first argument of `stops` is not a test function, then `stops(t, a)` returns the number 2.

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

   If the first argument of `stops` is a test function and, furthermore, the evaluation of $n(a)$ terminates, then `stops(t, a)` returns the number 1.

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

   If the first argument of `stops` is a test function but the evaluation of $n(a)$ diverges, then `stops(t, a)` returns the number 0.

If there was a *Python* function `stops` that did satisfy the specification given above, then the halting problem for *Python* would be decidable.

Theorem 3 (Alan Turing, 1936) *The halting problem is undecidable.*

Proof: In order to prove the undecidabilty of the halting problem we have to show that there can be no function `stops` satisfying the specification given above. This calls for an indirect proof also known as *proof by contradiction*. We will therefore assume that a function `stops` solving the halting problem does exist and we will then show that this assumption leads to a contradiction. This contradiction will leave us with the conclusion that there can be no function `stops` that satisfies the specification given above and that, therefore, the halting problem is not solvable.

In order to proceed, let us assume that a *Python* function `stops` satisfying the specification given above exists and let us define the string *turing* as shown in Figure 5.2 below.

```
turing = """def alan(x):
                result = stops(x, x)
                if result == 1:
                    while True:
                        print("... looping ...")
                return result
         """
```

Figure 5.2: Definition of the string *turing*.

Given this definition it is easy to check that *turing* is, indeed, a test function with the name “`alan`’, that is we have

$$turing \in TF \wedge \texttt{name}(turing) = \texttt{alan}.$$

Therefore, we can use the string *turing* as the first argument of the function `stops`. Let us think about the following expression:

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

Since we have already noted that *turing* is test function, according to the specification of the function `stops` there are only two cases left:

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

Let us consider these cases in turn.

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

   According to the specification of `stops` we should then have

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

   Let us check whether this is true. In order to do this, we have to check what happens when the expression

   ```
   alan(turing)
   ```

   is evaluated:

   (a) Since we have assumed for this case that the expression $\texttt{stops}(turing, turing)$ yields 0, in line 2, the variable `result` is assigned the value 0.

   (b) Line 3 now tests whether `result` is 1. Of course, this test fails. Therefore, the block of the `if`-statement is not executed.

   (c) Finally, in line 8 the value of the variable `result` is returned.

   All in all we see that the call of the function `alan` does terminate with the argument *turing*. However, this is the opposite of what the function `stops` has claimed.

   Therefore, this case has lead us to a contradiction.

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

   According to the specification of `stops` we should then have

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

   i.e. the evaluation of $\texttt{alan}(turing) \downarrow$ should terminate.

   Again, let us check in detail whether this is true.

   (a) Since we have assumed for this case that the expression $\texttt{stops}(turing, turing)$ yields 1, in line 2, the variable `result` is assigned the value 1.

   (b) Line 3 now tests whether `result` is 1. Of course, this time the test succeeds. Therefore, the block of the `if`-statement <u>is</u> executed.

   (c) However, this block contains an obvious infinite loop. Therefore, the evaluation of $\texttt{alan}(turing)$ <u>diverges</u>. But this contradicts the specification of `stops`!

   Therefore, the second case also leads to a contradiction.

As we have obtained contradictions in both cases, the assumption that there is a function `stops` that solves the halting problem is refuted. □

Remark: The proof of the fact that the halting problem is undecidable was given 1936 by Alan Turing (1912 – 1954) [Tur36]. Of course, Turing did not solve the problem for *Python* but rather for the so called *Turing machines*. A *Turing maschine* can be interpreted as a formal description of an algorithm. Therefore, Turing has shown, that there is no algorithm that is able to decide whether some given algorithm will always terminate.

Remark: At this point you might wonder whether there might be another programming language that is more powerful so that programming in this more powerful language it would be possible to solve the halting problem. However, if you check the proof given for *Python* you will easily see that this proof can be adapted to any other programming language that is as least as powerful as *Python*.

Of course, if a programming language is very restricted, then it might be possible to check the halting problem for this weak programming language. But for any programming language that supports at least `while`-loops, `if`-statements, and the definition of procedures the argument given above shows that the halting problem is not solvable.

Exercise 7: Show that if the halting problem would be solvable, then it would be possible to write a program that checks whether there are infinitely many *twin primes*. A *twin prime* is pair of natural numbers $\langle p, p+2\rangle$ such that both p and $p+2$ are prime numbers. The *twin prime conjecture* is one of the oldest unsolved mathematical problems.

Exercise 8: A set X is called *countable* iff there is a function

$$f : \mathbb{N} \rightarrow X$$

such that for all $x \in X$ there is a $n \in \mathbb{N}$ such that x is the image of n under f:

$$\forall x \in X : \exists n \in \mathbb{N} : x = f(n).$$

Prove that the set $2^{\mathbb{N}}$, which is the set of all subsets of $\mathbb{N}$ is not countable.

Hint: Your proof should be similar to the proof that the halting problem is undecidable. Proceed as follows: Assume that there is a function f enumerating the subsets of $\mathbb{N}$, that is assume that

$$\forall x \in 2^{\mathbb{N}} : \exists n \in \mathbb{N} : x = f(n)$$

holds. Next, and this is the crucial step, define a set `Cantor` as follows:

$$\texttt{Cantor} := \bigl\{ n \in \mathbb{N} \mid n \notin f(n) \bigr\}.$$

Now try to derive a contradiction.

5.2 Undecidability of the Equivalence Problem

Unfortunately, the halting problem is not the only undecidable problem in computer science. Another important problem that is undecidable is the question whether two given functions always compute the same result. To state this more formally, we need the following definition.

Definition 4 ($\simeq$) *Assume n_1 and n_2 are the names of two Python functions that take arguments a_1, $\cdots$, a_k. Let us define*

$$n_1(a_1, \cdots, a_k) \simeq n_2(a_1, \cdots, a_k)$$

if and only if either of the following cases is true:

1. $n_1(a_1, \cdots, a_k)\uparrow \quad \wedge \quad n_2(a_1, \cdots, a_k)\uparrow$,

   that is both function calls diverge.

2. $\exists r : \Big(n_1(a_1, \cdots, a_k) \leadsto r \quad \wedge \quad n_2(a_1, \cdots, a_k) \leadsto r \Big)$

   that is both function calls terminate and compute the same result.

If $n_1(a_1, \cdots, a_k) \simeq n_2(a_1, \cdots, a_k)$ *holds, then the expressions* $n_1(a_1, \cdots, a_k)$ *and* $n_2(a_1, \cdots, a_k)$ *are called* partially equivalent. □

We are now ready to state the *equivalence problem*. A *Python* function `equal` solves the *equivalence problem* if it is defined as

```
equal := procedure(p1, p2, a) { ...  };
```

and, furthermore, satisfies the following specification:

1. $p_1 \notin \mathit{TF} \vee p_2 \notin \mathit{TF} \quad \Leftrightarrow \quad \texttt{equal}(p_1, p_2, a) \leadsto 2.$
2. If
   (a) $p_1 \in \mathit{TF} \wedge \texttt{name}(p_1) = n_1,$
   (b) $p_2 \in \mathit{TF} \wedge \texttt{name}(p_2) = n_2$ and
   (c) $n_1(a) \simeq n_2(a)$

   holds, then we must have:

   $$\texttt{equal}(p_1, p_2, a) \leadsto 1.$$
3. Otherwise we must have

   $$\texttt{equal}(p_1, p_2, a) \leadsto 0.$$

Theorem 5 (Henry Gordon Rice, 1953)
The equivalence problem is undecidable.

Proof: The proof is by contradiction. Therefore, assume that there is a function `equal` such that `equal` solves the equivalence problem. Assuming `equal` exists, we will then proceed to define a function `stops` that does solve the halting problem. Figure 5.3 shows how we construct the function `stops` that makes use of the function `equal`.

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

Figure 5.3: An implementation of the function `stops`.

Notice that in line 2 the function `equal` is called with a string that is test function with name `loop`. This test function has the following form:

```
loop := procedure(x) { while (true) {} };
```

Obviously, the function `loop` does never terminate. Therefore, if the argument p of `stops` is a test function with name n, the function `equal` will return 1 if $n(a)$ diverges, and will return 0 otherwise. But this implementation of `stops` would then solve the halting problem as for a given test function p with name n and argument a the function `stops` would return 1 if and only the evaluation of $n(a)$ terminates. As we have already proven that the halting problem is undecidable, there can be no function `equal` that solves the equivalence problem either. □

5.3 Concluding Remarks

Although, in general, we cannot decide whether a program terminates for a given input, this does not mean that we should not attempt to do so. After all, we only have proven that there is no procedure that can always check whether a given program will terminate. There might well exist a procedure for termination checking that works most of the time. Indeed, there are a number of systems that try to check whether a program will always terminate. For example, for *Prolog* programs, the article "*Automated Modular Termination Proofs for Real Prolog Programs*" [MGS96] describes a successful approach. The recent years have seen a lot of progress in this area. The article "Proving Program Termination" [CPR11] reviews these developments. However, as the recently developed systems rely on both *automatic theorem proving* and *Ramsey theory* they are quite out of the scope of this lecture.

5.4 Chapter Review

You should be able to solve the following exercises.

1. Define the halting problem.
2. Define the equivalence problem.
3. Define the notion of a countable set.
4. Prove that the set $2^{\mathbb{N}}$ is not countable.

5.5 Further Reading

The book "*Introduction to the Theory of Computation*" by Michael Sipser [Sip96] discusses the undecidability of the halting problem in section 4.2. It also covers many related undecidable problems. The exposition in the book of Sipser is based on Turing machines.

Another good book discussing undecidability is the book "*Introduction to Automata Theory, Languages, and Computation*" written by John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman [HMU06]. This book is the third edition of a classic text. In this book, the topic of undecidability is discussed in chapter 9.

Chapter 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 wir 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. $a \rightarrow b$ 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 möglicherweise 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.

[1] Die Seite `https://en.wikipedia.org/wiki/Transistor_count` gibt einen Überblick über die Komplexität moderner Prozessoren.

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 für die Züge sogenannte Fahrstraßen 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.

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.

[2] Verum ist das lateinische Wort für "wahr".
[3] Falsum ist das lateinische Wort für "falsch".

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:

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 <u>links</u> geklammert werden, nennen wir links-assoziativ.

   <u>Beachten</u> Sie, dass wir für diese Vorlesung vereinbaren, dass die Junktoren $\wedge$ und $\vee$ dieselbe Bindungsstärke haben. Das ist anders als in der 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 <u>rechts</u> 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 diesen 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 dies 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*

$$\mathcal{I} : \mathcal{P} \rightarrow \mathbb{B},$$

die jeder Aussage-Variablen $p \in \mathcal{P}$ einen Wahrheitswert $\mathcal{I}(p) \in \mathbb{B}$ zuordnet. $\diamond$

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 Formeln interpretieren zu können, benötigen wir eine Interpretation der Junktoren "$\neg$", "$\wedge$", "$\vee$", "$\rightarrow$" und "$\leftrightarrow$". Zu diesem Zweck definieren wir auf der Menge $\mathbb{B}$ der Wahrheits-Werte Funktionen $\textcircled{$\neg$}$, $\textcircled{$\wedge$}$, $\textcircled{$\vee$}$, $\textcircled{$\rightarrow$}$ und $\textcircled{$\leftrightarrow$}$, mit deren Hilfe wir die aussagenlogischen Junktoren interpretieren können:

1. $\ominus : \mathbb{B} \to \mathbb{B}$
2. $\bigcirc\!\!\!\!\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
3. $\bigcirc\!\!\!\!\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
4. $\bigcirc\!\!\!\!\!\to : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
5. $\bigcirc\!\!\!\!\!\leftrightarrow : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

Wir könnten die Funktionen $\ominus$, $\bigcirc\!\!\!\!\wedge$, $\bigcirc\!\!\!\!\vee$, $\bigcirc\!\!\!\!\!\to$ und $\bigcirc\!\!\!\!\!\leftrightarrow$ am einfachsten durch die folgende Tabelle (Tabelle 6.1) definieren:

p	q	$\ominus(p)$	$\bigcirc\!\!\!\!\vee(p,q)$	$\bigcirc\!\!\!\!\wedge(p,q)$	$\bigcirc\!\!\!\!\!\to(p,q)$	$\bigcirc\!\!\!\!\!\leftrightarrow(p,q)$
`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`

Table 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. $\widehat{\mathcal{I}}(\neg f) := \ominus\big(\widehat{\mathcal{I}}(f)\big)$ für alle $f \in \mathcal{F}$.
5. $\widehat{\mathcal{I}}(f \wedge g) := \bigcirc\!\!\!\!\wedge\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ für alle $f, g \in \mathcal{F}$.
6. $\widehat{\mathcal{I}}(f \vee g) := \bigcirc\!\!\!\!\vee\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ für alle $f, g \in \mathcal{F}$.
7. $\widehat{\mathcal{I}}(f \to g) := \bigcirc\!\!\!\!\!\to\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ für alle $f, g \in \mathcal{F}$.
8. $\widehat{\mathcal{I}}(f \leftrightarrow g) := \bigcirc\!\!\!\!\!\leftrightarrow\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ 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:

$$
\begin{aligned}
\mathcal{I}\Big((p \to q) \to (\neg p \to q) \to q\Big) &= \ominus\!\!\!\to\Big(\mathcal{I}((p \to q)), \mathcal{I}((\neg p \to q) \to q)\Big) \\
&= \ominus\!\!\!\to\Big(\ominus\!\!\!\to(\mathcal{I}(p),\mathcal{I}(q)), \mathcal{I}((\neg p \to q) \to q)\Big) \\
&= \ominus\!\!\!\to\Big(\ominus\!\!\!\to(\texttt{True},\texttt{False}), \mathcal{I}((\neg p \to q) \to q)\Big) \\
&= \ominus\!\!\!\to\Big(\texttt{False}, \mathcal{I}((\neg p \to q) \to q)\Big) \\
&= \texttt{True}
\end{aligned}
$$

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 78 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 79 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

Table 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. $\mathcal{I}(p \to q) = \ominus\!\!\!\to(\mathcal{I}(p),\mathcal{I}(q)) = \ominus\!\!\!\to(\texttt{True},\texttt{False}) = \texttt{False}$
3. $\mathcal{I}(\neg p \to q) = \ominus\!\!\!\to(\mathcal{I}(\neg p),\mathcal{I}(q)) = \ominus\!\!\!\to(\texttt{False},\texttt{False}) = \texttt{True}$
4. $\mathcal{I}((\neg p \to q) \to q) = \ominus\!\!\!\to(\mathcal{I}(\neg p \to q),\mathcal{I}(q)) = \ominus\!\!\!\to(\texttt{True},\texttt{False}) = \texttt{False}$
5. $\mathcal{I}((p \to q) \to (\neg p \to q) \to q) = \ominus\!\!\!\to(\mathcal{I}(p \to q),\mathcal{I}((\neg p \to q) \to q)) = \ominus\!\!\!\to(\texttt{False},\texttt{False}) = \texttt{True}$

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 von *Python* automatisieren lässt.

6.3.3 Extensionale und intensionale Interpretationen der Aussagenlogik

Die Interpretation der 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.

Obwohl der folgende Satz mit der extensionalen Implikation

"*Wenn* $3 \cdot 3 = 8$*, dann schneit es.*"

als wahr interpretiert wird, erscheint er in ausgesprochener Form sinnlos.

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 letzterer 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 die 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:

$$rep(\bot) := \texttt{'\textbackslash N\{down tack\}'}.$$

3. Eine aussagenlogische Variable $p \in \mathcal{P}$ repräsentieren wir durch sich selber:

$$rep(p) := p \quad \text{für alle } p \in \mathcal{P}.$$

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:

$$rep(\neg f) := \big(\texttt{'\textbackslash N\{not sign\}'}, rep(f)\big).$$

5. Sind f_1 und f_2 aussagenlogische Formeln, so repräsentieren wir $f_1 \wedge f_2$ mit Hilfe des Unicode-Zeichens `'\N{logical and}'`:

$$rep(f \vee g) := \big(\texttt{'\textbackslash N\{logical and\}'}, rep(f), rep(g)\big).$$

6. Sind f_1 und f_2 aussagenlogische Formeln, so repräsentieren wir $f_1 \vee f_2$ mit Hilfe des Unicode-Zeichens `'\N{logical or}'`:

$$rep(f \vee g) := \big(\texttt{'\textbackslash N\{logical or\}'}, rep(f), rep(g)\big).$$

7. Sind f_1 und f_2 aussagenlogische Formeln, so repräsentieren wir $f_1 \rightarrow f_2$ mit Hilfe des Unicode-Zeichens `'\N{rightwards arrow}'`:

$$rep(f \rightarrow g) := \big(\texttt{'\textbackslash N\{rightwards arrow\}'}, rep(f), rep(g)\big).$$

8. Sind f_1 und f_2 aussagenlogische Formeln, so repräsentieren wir $f_1 \leftrightarrow f_2$ mit Hilfe des Unicode-Zeichens `'\N{left right arrow}'`:

$$rep(f \leftrightarrow g) := \big(\texttt{'\textbackslash N\{left right arrow\}'}, rep(f), rep(g)\big).$$

Bei der Wahl der Repräsentation, mit der wir eine Formel in *Python* 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 allee 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 82 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.

```
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)
```

Figure 6.1: Auswertung einer aussagenlogischen Formel

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`.

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 wir wissen, ob die Variable f ein Element der Menge I ist, denn genau dann wird f 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 $g_1 \rightarrow g_2$ hat. In diesem Fall werten wir zunächst g_1 und g_2 unter der Belegung I aus und nutzen dann aus, dass die Formeln

   $$g_1 \rightarrow g_2 \quad \text{und} \quad \neg g_1 \vee g_2$$

   äquivalent sind.

8. In Zeile 11 führen wir die Auswertung einer Formel $f \leftrightarrow g$ darauf zurück, dass diese Formel genau dann wahr ist, wenn f und g denselben 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 Aaron, Bernard und Caine 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 Aaron schuldig ist, so hat er genau einen Komplizen.

   Diese Aussage zerlegen wir zunächst in zwei Teilaussagen:

   (a) Wenn Aaron schuldig ist, dann hat er mindestens einen Komplizen:

   $$f_2 := a \rightarrow b \vee c$$

   (b) Wenn Aaron schuldig ist, dann hat er höchstens einen Komplizen:

   $$f_3 := a \rightarrow \neg(b \wedge c)$$

3. Wenn Bernard unschuldig ist, dann ist auch Caine unschuldig:

   $$f_4 := \neg b \rightarrow \neg c$$

4. Wenn genau zwei schuldig sind, dann ist Caine einer von ihnen.

   Es ist nicht leicht zu sehen, wie sich diese Aussage 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 Caine nicht schuldig ist und wenn gleichzeitig Aaron und Bernard schuldig sind. Damit lautet die Formalisierung der obigen Aussage:

   $$f_5 := \neg(\neg c \wedge a \wedge b)$$

5. Wenn Caine unschuldig ist, ist Aaron schuldig.

   $$f_6 := \neg c \rightarrow a$$

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 Belegung 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 dieses Programm nun Zeile für Zeile.

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.

```
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) })
```

Figure 6.2: Programm zur Aufklärung des Einbruchs

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

$$(p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q$$

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

$$\models f. \qquad \diamond$$

Ist eine Formel f eine Tautologie, so sagen wir auch, dass f allgemeingültig ist.
Beispiele:

1. $\models p \vee \neg p$
2. $\models p \rightarrow p$
3. $\models p \wedge q \rightarrow p$
4. $\models p \rightarrow p \vee q$
5. $\models (p \rightarrow \bot) \;\leftrightarrow\; \neg p$
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 79 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. $\diamond$

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
$\models (p \rightarrow q) \leftrightarrow \neg p \vee q$		Elimination von $\rightarrow$
$\models (p \leftrightarrow q) \leftrightarrow (\neg p \vee q) \wedge (\neg q \vee p)$		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

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`

Table 6.3: Nachweis der ersten DeMorgan'schen Regel

Spalten in jeder Zeile dieselben 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:

$$M_\mathcal{I} := \bigl\{ p \in \mathcal{P} \mid \mathcal{I}(p) = \texttt{True} \bigr\}.$$

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 87 zeigt, dass wir diese Gleichungen unmittelbar in *Python* umsetzen können, wobei wir die letzten vier Fälle zusammengefasst haben, denn die Berechnung der Variablen verläuft in diesen Fällen analog.

```
def collectVars(f):
    "Collect all propositional variables occurring in the formula f."
    if isinstance(f, str):
        return { f }
    if f[0] in ['⊤', '⊥']:
        return set()
    if f[0] == '¬':
        return collectVars(f[1])
    return collectVars(f[1]) | collectVars(f[2])
```

Figure 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 88 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.

```
def tautology(f):
    "Check, whether the formula f is a tautology."
    P = collectVars(f)
    A = power.allSubsets(P)
    if { evaluate(F, I) for I in A } == { True }:
        return True
    else:
        return [I for I in A if not evaluate(F, I)][0]
```

Figure 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}{lll}
 & (p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q & (\text{Elimination von } \rightarrow) \\
\Leftrightarrow & (\neg p \vee q) \rightarrow (\neg p \rightarrow q) \rightarrow q & (\text{Elimination von } \rightarrow) \\
\Leftrightarrow & (\neg p \vee q) \rightarrow (\neg\neg p \vee q) \rightarrow q & (\text{Elimination der Doppelnegation}) \\
\Leftrightarrow & (\neg p \vee q) \rightarrow (p \vee q) \rightarrow q & (\text{Elimination von } \rightarrow) \\
\Leftrightarrow & \neg(\neg p \vee q) \vee ((p \vee q) \rightarrow q) & (\text{DeMorgan}) \\
\Leftrightarrow & (\neg\neg p \wedge \neg q) \vee ((p \vee q) \rightarrow q) & (\text{Elimination der Doppelnegation}) \\
\Leftrightarrow & (p \wedge \neg q) \vee ((p \vee q) \rightarrow q) & (\text{Elimination von } \rightarrow) \\
\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 $\mathcal{K}$. $\diamond$

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. $\diamond$

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

$$\begin{aligned} & (F_1 \wedge \cdots \wedge F_m) \vee (G_1 \wedge \cdots \wedge G_n) \\ \leftrightarrow \quad & (F_1 \vee G_1) \wedge \cdots \wedge (F_1 \vee G_n) \;\wedge\; \cdots \;\wedge\; (F_m \vee G_1) \wedge \cdots \wedge (F_m \vee G_n) \end{aligned}$$

   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

$$(p \rightarrow q) \rightarrow (\neg p \rightarrow \neg q).$$

1. Da die Formel den Junktor "$\leftrightarrow$" nicht enthält, ist im ersten Schritt nichts zu tun.

2. Die Elimination von "$\rightarrow$" 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

$$\{\{p, \neg q\}\}.$$

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 $f = g_1 \to g_2$ eliminieren wir den Junktor "$\leftrightarrow$" aus den Formeln g_1 und g_2:

   $$\texttt{elimBiconditional}(g_1 \to g_2) = \texttt{elimBiconditional}(g_1) \to \texttt{elimBiconditional}(g_2).$$

7. Hat f die Form $f = g_1 \leftrightarrow g_2$, so benutzen wir die Äquivalenz

   $$(g_1 \leftrightarrow g_2) \leftrightarrow \big((g_1 \to g_2) \wedge (g_2 \to g_1)\big).$$

   Das führt auf die Gleichung:

   $$\texttt{elimBiconditional}(g_1 \leftrightarrow g_2) = \texttt{elimBiconditional}\big((g_1 \to g_2) \wedge (g_2 \to g_1)\big).$$

   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)
```

Figure 6.5: Elimination von $\leftrightarrow$

Abbildung 6.5 auf Seite 94 zeigt die Implementierung der Funktion `elimBiconditional`.

1. In Zeile 3 prüft der Funktions-Aufruf `isinstance`$(f,$`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 95 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.

```
def elimConditional(f):
    "Eliminate the logical operator '→' from f."
    if isinstance(f, str):
        return f
    if f[0] == '⊤':
        return f
    if f[0] == '⊥':
        return f
    if f[0] == '→':
        g, h = f[1:]
        ge   = elimConditional(g)
        he   = elimConditional(h)
        return ('∨', ('¬', ge), he)
    if f[0] == '¬':
        g  = f[1]
        ge = elimConditional(g)
        return ('¬', ge)
    else:
        op, g, h = f
        ge       = elimConditional(g)
        he       = elimConditional(h)
        return (op, ge, he)
```

Figure 6.6: Elimination von $\rightarrow$

Als nächstes zeigen wir die Funktionen zur Berechnung der Negations-Normalform. Abbildung 6.7 auf Seite 97 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. $\texttt{nnf}(p) = p \quad$ 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. $$\begin{array}{cl} & \texttt{neg}\big(f_1 \wedge f_2\big) \\ = & \texttt{nnf}\big(\neg(f_1 \wedge f_2)\big) \\ = & \texttt{nnf}\big(\neg f_1 \vee \neg f_2\big) \\ = & \texttt{nnf}\big(\neg f_1\big) \vee \texttt{nnf}\big(\neg f_2\big) \\ = & \texttt{neg}(f_1) \vee \texttt{neg}(f_2). \end{array}$$
   Also haben wir:
   $$\texttt{neg}\big(f_1 \wedge f_2\big) = \texttt{neg}(f_1) \vee \texttt{neg}(f_2).$$
6. $$\begin{array}{cl} & \texttt{neg}\big(f_1 \vee f_2\big) \\ = & \texttt{nnf}\big(\neg(f_1 \vee f_2)\big) \\ = & \texttt{nnf}\big(\neg f_1 \wedge \neg f_2\big) \\ = & \texttt{nnf}\big(\neg f_1\big) \wedge \texttt{nnf}\big(\neg f_2\big) \\ = & \texttt{neg}(f_1) \wedge \texttt{neg}(f_2). \end{array}$$
   Also haben wir:
   $$\texttt{neg}\big(f_1 \vee f_2\big) = \texttt{neg}(f_1) \wedge \texttt{neg}(f_2).$$

Die in Abbildung 6.7 auf Seite 97 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 ('∨', neg(g), neg(h))
    if f[0] == '∨':
        g, h = f[1:]
        return ('∧', neg(g), neg(h))
```

Figure 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:
$$\texttt{cnf}(f) := \big\{\{f\}\big\} \quad \text{falls } f \in \mathcal{P}.$$
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:
$$\texttt{cnf}(\bot) := \big\{\{\}\big\}.$$
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:
$$\texttt{cnf}(\neg p) := \big\{\{\neg p\}\big\} \quad \text{falls } p \in \mathcal{P}.$$
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:
$$\begin{array}{ll} & g \vee h \\ \Leftrightarrow & (k_1 \wedge \cdots \wedge k_m) \vee (l_1 \wedge \cdots \wedge l_n) \\ \Leftrightarrow & (k_1 \vee l_1) \quad \wedge \quad \cdots \quad \wedge \quad (k_m \vee l_1) \quad \wedge \\ & \quad\vdots \qquad\qquad\qquad\qquad\quad \vdots \\ & (k_1 \vee l_n) \quad \wedge \quad \cdots \quad \wedge \quad (k_m \vee l_n) \\ \Leftrightarrow & \big\{k_i \vee l_j : i \in \{1, \cdots, m\}, j \in \{1, \cdots, n\}\big\} \end{array}$$
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 99 zeigt die Implementierung der Funktion `cnf`. (Der Name `cnf` ist die Abkürzung von *conjunctive normal form*.)

```
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) }
```

Figure 6.8: Berechnung der konjunktiven Normalform

Zum Abschluss zeigen wir in Abbildung 6.9 auf Seite 100 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.
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.

Aufgabe 9: 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.

```
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)
```

Figure 6.9: Normalisierung einer Formel

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. $\neg r \wedge (q \vee p \rightarrow r) \rightarrow \neg q \wedge \neg p$.

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

$$\models f_1 \wedge \cdots \wedge f_n \rightarrow g$$

gilt. Es gibt verschiedene Möglichkeiten, diese Frage zu beantworten. Ein Verfahren kennen wir schon: Zunächst überführen wir die Formel $f_1 \wedge \cdots \wedge f_n \rightarrow g$ in konjunktive Normalform. Wir erhalten dann eine Menge $\{k_1, \cdots, k_m\}$ von Klauseln, deren Konjunktion zu der Formel

$$f_1 \wedge \cdots \wedge f_n \rightarrow g$$

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 $p \rightarrow r$ aus den beiden Formeln $p \rightarrow q$ und $q \rightarrow r$ folgt. Wir bilden also die konjunktive Normalform der Formel

$$h := (p \rightarrow q) \wedge (q \rightarrow r) \rightarrow p \rightarrow r$$

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
$\dfrac{f \qquad f \to g}{g}$	$\dfrac{\neg g \qquad f \to g}{\neg f}$	$\dfrac{\neg f \qquad f \to g}{\neg g}$

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 $\models f_1 \wedge \cdots \wedge f_n \to k$ 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:*

$$\frac{k_1 \cup \{p\} \quad \{\neg p\} \cup k_2}{k_1 \cup k_2}. \qquad \diamond$$

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 $p \rightarrow r$ aus $p \rightarrow q$ und $q \rightarrow r$ folgt. Dazu überführen wir zunächst alle Formeln in Klauseln:

$$\texttt{cnf}(p \rightarrow q) = \big\{\{\neg p, q\}\big\}, \quad \texttt{cnf}(q \rightarrow r) = \big\{\{\neg q, r\}\big\}, \quad \texttt{cnf}(p \rightarrow r) = \big\{\{\neg p, r\}\big\}.$$

Wir haben also $M = \big\{\ \{\neg p, q\},\ \{\neg q, r\}\ \big\}$ 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:*

$$\textit{Wenn } \{k_1, \cdots, k_n\} \vdash k \textit{ gilt, dann gilt auch} \models k_1 \wedge \cdots \wedge k_n \rightarrow k.$$

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

   $$\models k_1 \wedge \cdots \wedge k_i \wedge \cdots \wedge k_n \rightarrow k_i$$

   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

$$\mathcal{I}(g \vee h) = \texttt{True}. \ \surd$$

(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

$$\mathcal{I}(g \vee h) = \texttt{True}. \ \surd$$

□

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. $\diamond$

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 $\textsc{Ali}$*, so schreibt sich das formal als*

$$M \models \bot \quad \text{g.d.w.} \quad \forall \mathcal{I} \in \textsc{Ali} : \exists g \in M : \mathcal{I}(g) = \texttt{False}. \qquad \diamond$$

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/Logic/blob/master/Python/Completeness.ipynb`

im Netz. Die Grundidee bei diesem Programm besteht darin, dass wir versuchen, aus einer gegebenen Menge M von Klauseln <u>alle</u> 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.

```
def complement(l):
    "Compute the complement of the literal l."
    if isinstance(l, str):  # l is a propositional variable
        return ('¬', l)
    else:                   # l = ('¬', 'p')
    return l[1]

def extractVariable(l):
    "Extract the variable of the literal l."
    if isinstance(l, str):  # l is a propositional variable
        return l
    else:                   # l = ('¬', 'p')
        return l[1]

def collectVariables(M):
    "Return the set of all variables occurring in M."
    return { extractVariable(l) for C in M
                                for l in C
           }

def cutRule(C1, C2):
    "Return the set of all clauses that can be deduced by the cut rule from c1 and c2."
    return { C1 - {l} | C2 - {complement(l)} for l in C1
                                             if complement(l) in C2
           }
```

Figure 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:
$$C_1 \cup \{l\} \in S \;\land\; C_2 \cup \{\overline{l}\} \in S \;\Rightarrow\; C_1 \cup C_2 \in S$$

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

```
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
```

Figure 6.11: Die Funktion `saturate`

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.

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 <u>neue</u> 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.

```
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
```

Figure 6.12: Die Funktion `findValuation`

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.

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 $\{$`True`, `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 $\neg$`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

$$\mathcal{I}(\mathtt{p}) = \begin{cases} \mathtt{True} & \text{falls } \;\; \mathtt{p} \in \mathtt{Literals} \\ \mathtt{False} & \text{falls } \neg\mathtt{p} \in \mathtt{Literals} \end{cases}$$

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 $\neg$`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 $\mathcal{I}(p_i) = \mathtt{True}$ und falls $l_i = \neg p_i$ gilt, so haben wir $\mathcal{I}(p_i) = \mathtt{False}$. Nehmen wir nun weiter an, dass eine Klausel C in der Menge `Clauses` existiert, so dass

   $$C\backslash\{p\} \subseteq \{\,\overline{l_1}, \cdots, \overline{l_n}\,\} \quad \text{und} \quad \mathtt{p} \in C$$

   gilt. Wenn $\mathcal{I}(C) = \mathtt{True}$ gelten soll, dann muss $\mathcal{I}(\mathtt{p}) = \mathtt{True}$ gelten, denn nach Konstruktion von $\mathcal{I}$ gilt

   $$\mathcal{I}\big(\overline{l_i}\big) = \mathtt{False} \quad \text{für alle } i \in \{1, \cdots, n\}$$

   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 $D \in$ `Clauses`, in der nur die Variablen $p_1, \cdots, p_n$ vorkommen,

$$\mathcal{I}(D) = \mathtt{True}$$

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 $C \in \texttt{Clauses}$ mit

$$C \subseteq \left\{\overline{l} \mid l \in \texttt{Literals}\right\} \cup \{p_{n+1}\} \quad \text{und} \quad p_{n+1} \in C$$

geben kann und damit tritt der hier untersuchte Fall gar nicht auf.

ii. Es gibt keine Klausel C in der Menge `Clauses`, so dass

$$C \subseteq \left\{\overline{l} \mid l \in \texttt{Literals}\right\} \cup \{p_{n+1}\} \quad \text{und} \quad p_{n+1} \in C$$

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 $C \in \texttt{Clauses}$ nur eine endliche Anzahl von Variablen enthält, haben wir insgesamt gezeigt, dass für alle diese Klauseln $\mathcal{I}(C) = \texttt{True}$ gilt. $\square$

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

$$K_3 = \{\{p\}, \{\neg q\}, \{\neg p\}\}.$$

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:

$$\texttt{unitCut}(K, l) = \left\{ C \backslash \{\overline{l}\} \;\middle|\; C \in K \right\}.$$

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

$$K = \{\{p, q, \neg r\}, \{p\}\} \cup M.$$

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:

$$\texttt{subsume}(K, l) := \big(K \backslash \{C \in K \mid l \in C\}\big) \cup \{\{l\}\} = \{C \in K \mid l \notin C\} \cup \{\{l\}\}.$$

In der obigen Definition muss $\{l\}$ in das Ergebnis eingefügt werden, weil die Menge $\{C \in K \mid l \notin C\}$ 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 $K \cup \{\{p\}\}$ oder $K \cup \{\{\neg p\}\}$ 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 $K \cup \{\{p\}\}$ erfüllbar, andernfalls ist $K \cup \{\{\neg p\}\}$ erfüllbar.

Da K sowohl eine Teilmenge von $K \cup \{\{p\}\}$ als auch von $K \cup \{\{\neg p\}\}$ 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

$$K_1 := K \cup \{\{p\}\} \quad \text{und} \quad K_2 := K \cup \{\{\neg p\}\}$$

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

   $$K \cup \{\{p\}\}$$

   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

   $$K_1 := \texttt{reduce}(K_0,p) = \Big\{ \{r,\neg t\},\ \{r,s\},\ \{\neg r,q\},\ \{\neg q,s,\neg r\},\ \{\neg r,\neg s\},\ \{\neg s\},\ \{p\} \Big\}.$$

   Die Klausel-Menge K_1 enthält die Unit-Klausel $\{\neg s\}$, so dass wir als nächstes mit dieser Klausel reduzieren können:

   $$K_2 := \texttt{reduce}(K_1,\neg s) = \Big\{ \{r,\neg t\},\ \{r\},\ \{\neg r,q\},\ \{\neg q,\neg r\},\ \{\neg s\},\ \{p\} \Big\}.$$

   Hier haben wir nun die neue Unit-Klausel $\{r\}$, mit der wir weiter reduzieren:

   $$K_3 := reduce(K_2,r) = \Big\{ \{r\},\ \{q\},\ \{\neg q\},\ \{\neg s\},\ \{p\} \Big\}$$

   Da K_3 die Unit-Klausel $\{q\}$ enthält, reduzieren wir jetzt mit q:

   $$K_4 := reduce(K_2,q) = \Big\{ \{r\},\ \{q\},\ \{\},\ \{\neg s\},\ \{p\} \Big\}.$$

   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

   $$K_6 = \mathit{reduce}(K_5, \neg p) = \Big\{ \{q,s\},\ \{r,s\},\ \{\neg s\},\ \{\neg q,s\},\ \{\neg r,\neg s\},\ \{\neg p\} \Big\}.$$

   Die Menge K_6 enthält die Unit-Klausel $\{\neg s\}$. Wir bilden daher

   $$K_7 = \mathit{reduce}(K_6, \neg s) = \Big\{ \{q\},\ \{r\},\ \{\neg s\},\ \{\neg q\},\ \{\neg p\} \Big\}.$$

   Die Menge K_7 enthält die neue Unit-Klausel $\{q\}$, mit der wir als nächstes reduzieren:

   $$K_8 = \mathit{reduce}(K_7, q) = \Big\{ \{q\},\ \{r\},\ \{\neg s\},\ \{\},\ \{\neg p\} \Big\}.$$

   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 119 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`.

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.

```
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 })
```

Figure 6.13: Die Funktion `solve`

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

   $$\texttt{S} \cup \Big\{\{\overline{\texttt{l}}\}\Big\}$$

   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 120 gezeigt.

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.

```
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
```

Figure 6.14: Die Funktion `saturate`

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 $\mathit{reduce}(S,l)$, 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.

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 121 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 $C \in$

```
def reduce(Clauses, l):
    lBar = complement(l)
    return   { C - { lBar } for C in Clauses if lBar in C }         \
           | { C for C in Clauses if lBar not in C and l not in C }  \
           | { frozenset({l}) }
```

Figure 6.15: Die Funktion `reduce`

`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.

```
def selectLiteral(Clauses, Forbidden):
    Variables = { extractVariable(l) for C in Clauses for l in C } - Forbidden
    return arb(Variables)

def arb(S):
    "Return some member from the set S."
    for x in S:
        return x
```

Figure 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

Aufgabe 10: Die Klausel-Menge M sei wie folgt gegeben:

$$M := \{ \{r,p,s\}, \{r,s\}, \{q,p,s\}, \{\neg p, \neg q\}, \{\neg p, s, \neg r\}, \{p, \neg q, r\},$$
$$\{\neg r, \neg s, q\}, \{p,q,r,s\}, \{r, \neg s, q\}, \{\neg r, s, \neg q\}, \{s, \neg r\} \}$$

Überprüfen Sie mit dem Verfahren von Davis und Putnam, ob die Menge M widersprüchlich ist. ◇

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 122 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.

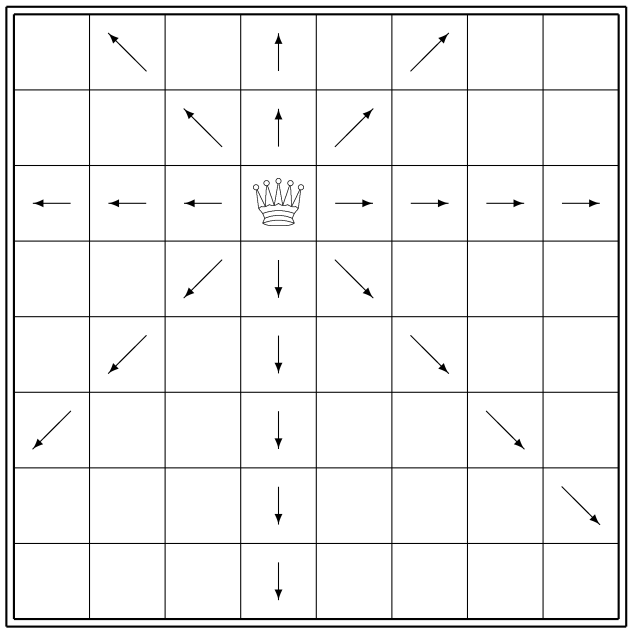

Figure 6.17: Das 8-Damen-Problem

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 123 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.

```
def var(row, col):
    return 'Q' + str(row) + ',' + str(col) + ')'
```

Figure 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)

Figure 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.

```
def atMostOne(S):
    return { frozenset({('¬',p), ('¬', q)}) for p in S
                                                for q in S
                                                if  p != q
             }
```

Figure 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.

```
def atMostOneInRow(row, n):
    return atMostOne({ var(row, col) for col in range(1,n+1) })
```

Figure 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

$$\mathtt{Q}(1,c) \vee \mathtt{Q}(2,c) \vee \mathtt{Q}(3,c) \vee \mathtt{Q}(4,c) \vee \mathtt{Q}(5,c) \vee \mathtt{Q}(6,c) \vee \mathtt{Q}(7,c) \vee \mathtt{Q}(8,c).$$

Schreiben wir diese Formel in der Mengenschreibweise als Menge von Klauseln, so erhalten wir

$$\big\{\{\mathtt{Q}(1,c), \mathtt{Q}(2,c), \mathtt{Q}(3,c), \mathtt{Q}(4,c), \mathtt{Q}(5,c), \mathtt{Q}(6,c), \mathtt{Q}(7,c), \mathtt{Q}(8,c)\}\big\}.$$

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.

```
def oneInColumn(col, n):
    return { frozenset({ var(row, col) for row in range(1,n+1) }) }
```

Figure 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.

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

$$r - c = 0.$$

```
def atMostOneInRisingDiagonal(k, n):
    S = { var(row, col) for row in range(1, n+1)
                        for col in range(1, n+1)
                        if  row + col == k
        }
    return atMostOne(S)
```

Figure 6.23: Die Funktion `atMostOneInUpperDiagonal`

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.

```
def atMostOneInFallingDiagonal(k, n):
    S = { var(row, col) for row in range(1, n+1)
                        for col in range(1, n+1)
                        if  row - col == k
        }
    return atMostOne(S)
```

Figure 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

$$\texttt{allClauses}(n)$$

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.

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.

```
def allClauses(n):
    All = [ atMostOneInRow(row, n)          for row in range(1, n+1)        ] \
        + [ atMostOneInFallingDiagonal(k, n) for k in range(3, (2*n-1)+1)    ] \
        + [ atMostOneInRisingDiagonal(k, n)  for k in range(-(n-2), (n-2)+1) ] \
        + [ oneInColumn(col, n)             for col in range(1, n+1)        ]
    return { clause for S in All for clause in S }
```

Figure 6.25: Die Funktion `allClauses`

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!')
```

Figure 6.26: Die Funktion `queens`

Die durch den Aufruf `solve`$(\textit{Clauses}, \{\})$ 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.

```
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)
```

Figure 6.27: Die Funktion `printBoard`()

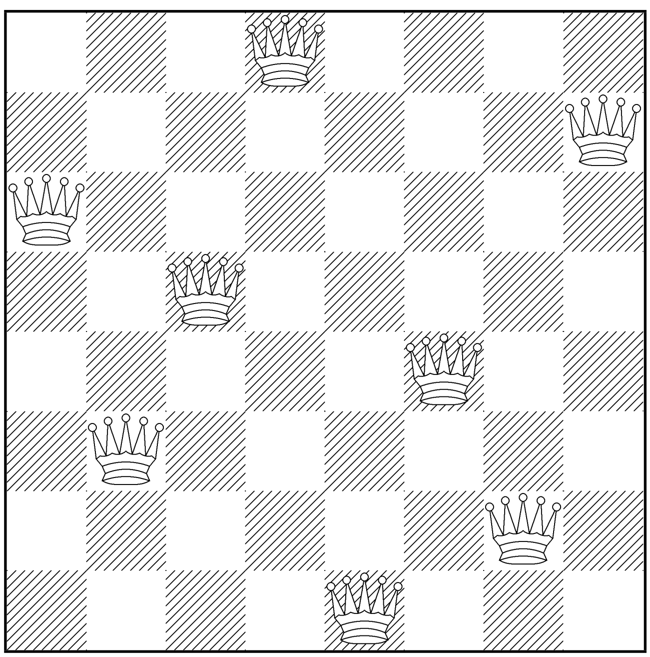

Figure 6.28: Eine Lösung des 8-Damen-Problems

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?

Chapter 7

Prädikatenlogik

In der Aussagenlogik haben wir die Verknüpfung von elementaren Aussagen mit Junktoren untersucht. Die Prädikatenlogik untersucht zusätzlich auch die Struktur dieser elementaren Aussagen. Dazu werden in der Prädikatenlogik die folgenden zusätzlichen Begriffe eingeführt:

1. Als Bezeichnungen für Objekte werden Terme verwendet.
2. Diese Terme werden aus Variablen und Funktions-Zeichen zusammengesetzt:
$$\mathit{vater}(x), \quad \mathit{mutter}(\mathit{isaac}), \quad x+7, \quad \cdots.$$
3. Verschiedene Objekte werden durch Prädikats-Zeichen in Relation gesetzt:
$$\mathit{istBruder}\big(\mathit{albert}, \mathit{vater}(\mathit{bruno})\big), \quad x+7 < x \cdot 7, \quad n \in \mathbb{N}, \quad \cdots.$$
Die dabei entstehenden Formeln werden als atomare Formeln bezeichnet.
4. Atomare Formeln lassen sich durch aussagenlogische Junktoren verknüpfen:
$$x > 1 \rightarrow x + 7 < x \cdot 7.$$
5. Schließlich werden Quantoren eingeführt, um zwischen existentiell und universell quantifizierten Variablen unterscheiden zu können:
$$\forall x \in \mathbb{R} : \exists n \in \mathbb{N} : x < n.$$

Wir werden im nächsten Abschnitt die Syntax der prädikatenlogischen Formeln festlegen und uns dann im darauf folgenden Abschnitt mit der Semantik dieser Formeln beschäftigen.

7.1 Syntax der Prädikatenlogik

Zunächst definieren wir den Begriff der Signatur. Inhaltlich ist das nichts anderes als eine strukturierte Zusammenfassung von Variablen, Funktions- und Prädikats-Zeichen zusammen mit einer Spezifikation der Stelligkeit dieser Zeichen.

Definition 26 (Signatur) *Eine Signatur ist ein 4-Tupel*
$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle,$$
für das Folgendes gilt:

1. *$\mathcal{V}$ ist die Menge der Variablen.*
2. *$\mathcal{F}$ ist die Menge der Funktions-Zeichen.*
3. *$\mathcal{P}$ ist die Menge der Prädikats-Zeichen.*
4. *arity ist eine Funktion, die jedem Funktions- und jedem Prädikats-Zeichen seine Stelligkeit zuordnet:*

   $$\textit{arity} : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}.$$

   Wir sagen, dass das Funktions- oder Prädikats-Zeichen f ein n-stelliges Zeichen ist, falls $\textit{arity}(f) = n$ gilt.
5. *Da wir in der Lage sein müssen, Variablen, Funktions- und Prädikats-Zeichen unterscheiden zu können, vereinbaren wir, dass die Mengen $\mathcal{V}$, $\mathcal{F}$ und $\mathcal{P}$ paarweise disjunkt sein müssen:*

   $$\mathcal{V} \cap \mathcal{F} = \{\}, \quad \mathcal{V} \cap \mathcal{P} = \{\}, \quad \text{und} \quad \mathcal{F} \cap \mathcal{P} = \{\}. \qquad \diamond$$

Als Bezeichner für Objekte verwenden wir Ausdrücke, die aus Variablen und Funktions-Zeichen aufgebaut sind. Solche Ausdrücke nennen wir Terme. Formal werden diese wie folgt definiert.

Definition 27 (Terme, $\mathcal{T}_\Sigma$) *Ist $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \textit{arity} \rangle$ eine Signatur, so definieren wir die Menge der Σ-Terme $\mathcal{T}_\Sigma$ induktiv:*

1. *Für jede Variable $x \in \mathcal{V}$ gilt $x \in \mathcal{T}_\Sigma$.*
2. *Ist $f \in \mathcal{F}$ ein n-stelliges Funktions-Zeichen und sind $t_1, \cdots, t_n \in \mathcal{T}_\Sigma$, so gilt auch*

   $$f(t_1, \cdots, t_n) \in \mathcal{T}_\Sigma.$$

   Falls $c \in \mathcal{F}$ ein 0-stelliges Funktions-Zeichen ist, lassen wir auch die Schreibweise c anstelle von $c()$ zu. In diesem Fall nennen wir c eine Konstante. $\diamond$

Beispiel: Es sei

1. $\mathcal{V} := \{x, y, z\}$ die Menge der Variablen,
2. $\mathcal{F} := \{0, 1, +, -, *\}$ die Menge der Funktions-Zeichen,
3. $\mathcal{P} := \{=, \leq\}$ die Menge der Prädikats-Zeichen,
4. $\textit{arity} := \big\{\langle 0,0\rangle, \langle 1,0\rangle, \langle +,2\rangle, \langle -,2\rangle, \langle *,2\rangle, \langle =,2\rangle, \langle \leq,2\rangle\big\}$,

   gibt die Stelligkeit der Funktions- und Prädikats-Zeichen an und
5. $\Sigma_{\text{arith}} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \textit{arity} \rangle$ sei eine Signatur.

Dann können wir wie folgt Σ_{arith}-Terme konstruieren:

1. $x, y, z \in \mathcal{T}_{\Sigma_{\text{arith}}}$,

   denn alle Variablen sind auch Σ_{arith}-Terme.
2. $0, 1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$,

   denn 0 und 1 sind 0-stellige Funktions-Zeichen.

3. $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
   denn es gilt $0 \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $x \in \mathcal{T}_{\Sigma_{\text{arith}}}$ und $+$ ist ein 2-stelliges Funktions-Zeichen.
4. $*(+(0, x), 1) \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
   denn $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$ und $*$ ist ein 2-stelliges Funktions-Zeichen.

In der Praxis werden wir für bestimmte zweistellige Funktionen eine Infix-Schreibweise verwenden. Diese ist dann als Abkürzung für die oben definierte Darstellung zu verstehen. $\diamond$

Als nächstes definieren wir den Begriff der atomaren Formeln. Darunter verstehen wir solche Formeln, die man nicht in kleinere Formeln zerlegen kann: Atomare Formeln enthalten also weder Junktoren noch Quantoren.

Definition 28 (Atomare Formeln, $\mathcal{A}_\Sigma$) *Gegeben sei eine Signatur $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \textit{arity} \rangle$. Die Menge der atomaren Σ-Formeln $\mathcal{A}_\Sigma$ wird wie folgt definiert: Ist $p \in \mathcal{P}$ ein n-stelliges Prädikats-Zeichen und sind n Σ-Terme $t_1, \cdots, t_n$ gegeben, so ist $p(t_1, \cdots, t_n)$ eine atomaren Σ-Formel:*

$$p(t_1, \cdots, t_n) \in \mathcal{A}_\Sigma.$$

Falls p ein 0-stelliges Prädikats-Zeichen ist, dann schreiben wir auch p anstelle von $p()$. In diesem Fall nennen wir p eine Aussage-Variable. $\diamond$

Beispiel: Setzen wir das letzte Beispiel fort, so können wir sehen, dass

$$=(*(+(0, x), 1), 0)$$

eine atomare Σ_{arith}-Formel ist. Beachten Sie, dass wir bisher noch nichts über den Wahrheitswert von solchen Formeln ausgesagt haben. Die Frage, wann eine Formel als wahr oder falsch gelten soll, wird erst im nächsten Abschnitt untersucht. $\diamond$

Bei der Definition der prädikatenlogischen Formeln ist es notwendig, zwischen sogenannten gebundenen und freien Variablen zu unterscheiden. Wir führen diese Begriffe zunächst informal mit Hilfe eines Beispiels aus der Analysis ein. Wir betrachten die folgende Gleichung:

$$\int_0^x y \cdot t \, dt = \frac{1}{2} \cdot x^2 \cdot y$$

In dieser Gleichung treten die Variablen x und y frei auf, während die Variable t durch das Integral gebunden wird. Damit meinen wir folgendes: Wir können in dieser Gleichung für x und y beliebige Werte einsetzen, ohne dass sich an der Gültigkeit der Formel etwas ändert. Setzen wir zum Beispiel für x den Wert 2 ein, so erhalten wir

$$\int_0^2 y \cdot t \, dt = \frac{1}{2} \cdot 2^2 \cdot y$$

und diese Gleichung ist ebenfalls gültig. Demgegenüber macht es keinen Sinn, wenn wir für die gebundene Variable t eine Zahl einsetzen würden. Die linke Seite der entstehenden Gleichung wäre einfach undefiniert. Wir können für t höchstens eine andere Variable einsetzen. Ersetzen wir die Variable t beispielsweise durch u, so erhalten wir

$$\int_0^x y \cdot u \, du = \frac{1}{2} \cdot x^2 \cdot y$$

und das ist inhaltlich dieselbe Aussage wie oben. Das funktioniert allerdings nicht mit jeder Variablen. Setzen wir für t die Variable y ein, so erhalten wir

$$\int_0^x y \cdot y \, dy = \frac{1}{2} \cdot x^2 \cdot y.$$

Diese Aussage ist aber falsch! Das Problem liegt darin, dass bei der Ersetzung von t durch y die vorher freie Variable y gebunden wurde.

Ein ähnliches Problem erhalten wir, wenn wir für y beliebige Terme einsetzen. Solange diese Terme die Variable t nicht enthalten, geht alles gut. Setzen wir beispielsweise für y den Term x^2 ein, so erhalten wir

$$\int_0^x x^2 \cdot t \, dt = \frac{1}{2} \cdot x^2 \cdot x^2$$

und diese Formel ist gültig. Setzen wir allerdings für y den Term t^2 ein, so erhalten wir

$$\int_0^x t^2 \cdot t \, dt = \frac{1}{2} \cdot x^2 \cdot t^2$$

und diese Formel ist nicht mehr gültig.

In der Prädikatenlogik binden die Quantoren "$\forall$" (für alle) und "$\exists$" (es gibt) Variablen in ähnlicher Weise, wie der Integral-Operator "$\int \cdot \; \mathtt{d}t$" in der Analysis Variablen bindet. Die oben gemachten Ausführungen zeigen, dass es zwei verschiedene Arten von Variable gibt: freie Variablen und gebundene Variablen. Um diese Begriffe präzisieren zu können, definieren wir zunächst für einen Σ-Term t die Menge der in t enthaltenen Variablen.

Definition 29 ($\mathit{\mathbf{Var}}(t)$) *Ist $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$ eine Signatur und ist t ein Σ-Term, so definieren wir die Menge $\mathit{Var}(t)$ der Variablen, die in t auftreten, durch Induktion nach dem Aufbau des Terms:*

1. $\mathit{Var}(x) := \{x\}$ *für alle* $x \in \mathcal{V}$,
2. $\mathit{Var}\big(f(t_1, \cdots, t_n)\big) := \mathit{Var}(t_1) \cup \cdots \cup \mathit{Var}(t_n)$. $\diamond$

Definition 30 (Σ-Formel, $\mathbb{F}_\Sigma$, gebundene und freie Variablen, $\mathit{\mathbf{BV}}(F)$, $\mathit{\mathbf{FV}}(F)$)
Es sei $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$ eine Signatur. Die Menge der Σ-Formeln bezeichnen wir mit $\mathbb{F}_\Sigma$. Wir definieren diese Menge induktiv. Gleichzeitig definieren wir für jede Formel $F \in \mathbb{F}_\Sigma$ die Menge $\mathit{BV}(F)$ der in F gebunden auftretenden Variablen und die Menge $\mathit{FV}(F)$ der in F frei auftretenden Variablen.

1. *Es gilt $\bot \in \mathbb{F}_\Sigma$ und $\top \in \mathbb{F}_\Sigma$ und wir definieren*
   $$\mathit{FV}(\bot) := \mathit{FV}(\top) := \mathit{BV}(\bot) := \mathit{BV}(\top) := \{\}.$$
2. *Ist $F = p(t_1, \cdots, t_n)$ eine atomare Σ-Formel, so gilt $F \in \mathbb{F}_\Sigma$. Weiter definieren wir:*
   (a) $\mathit{FV}\big(p(t_1, \cdots, t_n)\big) := \mathit{Var}(t_1) \cup \cdots \cup \mathit{Var}(t_n)$.
   (b) $\mathit{BV}\big(p(t_1, \cdots, t_n)\big) := \{\}$.
3. *Ist $F \in \mathbb{F}_\Sigma$, so gilt $\neg F \in \mathbb{F}_\Sigma$. Weiter definieren wir:*
   (a) $\mathit{FV}\big(\neg F\big) := \mathit{FV}(F)$.
   (b) $\mathit{BV}\big(\neg F\big) := \mathit{BV}(F)$.
4. *Sind $F, G \in \mathbb{F}_\Sigma$ und gilt außerdem*
   $$\big(\mathit{FV}(F) \cup \mathit{FV}(G)\big) \cap \big(\mathit{BV}(F) \cup \mathit{BV}(G)\big) = \{\},$$
   so gilt auch

(a) $(F \wedge G) \in \mathbb{F}_\Sigma$,

(b) $(F \vee G) \in \mathbb{F}_\Sigma$,

(c) $(F \rightarrow G) \in \mathbb{F}_\Sigma$,

(d) $(F \leftrightarrow G) \in \mathbb{F}_\Sigma$.

Weiter definieren wir für alle Junktoren $\odot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$*:*

(a) $FV\big((F \odot G)\big) := FV(F) \cup FV(G)$.

(b) $BV\big((F \odot G)\big) := BV(F) \cup BV(G)$.

5. *Sei* $x \in \mathcal{V}$ *und* $F \in \mathbb{F}_\Sigma$ *mit* $x \notin BV(F)$. *Dann gilt:*

(a) $(\forall x\colon F) \in \mathbb{F}_\Sigma$.

(b) $(\exists x\colon F) \in \mathbb{F}_\Sigma$.

Weiter definieren wir

(a) $FV\big((\forall x\colon F)\big) := FV\big((\exists x\colon F)\big) := FV(F)\backslash\{x\}$.

(b) $BV\big((\forall x\colon F)\big) := BV\big((\exists x\colon F)\big) := BV(F) \cup \{x\}$.

Ist die Signatur Σ *aus dem Zusammenhang klar oder aber unwichtig, so schreiben wir auch* $\mathbb{F}$ *statt* $\mathbb{F}_\Sigma$ *und sprechen dann einfach von Formeln statt von* Σ*-Formeln.* ⋄

Bei der oben gegebenen Definition haben wir darauf geachtet, dass eine Variable nicht gleichzeitig frei und gebunden in einer Formel auftreten kann, denn durch eine leichte Induktion nach dem Aufbau der Formeln lässt sich zeigen, dass für alle $F \in \mathbb{F}_\Sigma$ folgendes gilt:

$$FV(F) \cap BV(F) = \{\}.$$

Beispiel: Setzen wir das oben begonnene Beispiel fort, so sehen wir, dass

$$(\exists x\colon \ \leq(+(y,x),y))$$

eine Formel aus $\mathbb{F}_{\Sigma_{\text{arith}}}$ ist. Die Menge der gebundenen Variablen ist $\{x\}$, die Menge der freien Variablen ist $\{y\}$. ⋄

Wenn wir Formeln immer in der oben definierten Präfix-Notation anschreiben würden, dann würde die Lesbarkeit unverhältnismäßig leiden. Zur Abkürzung vereinbaren wir, dass in der Prädikatenlogik dieselben Regeln zur Klammer-Ersparnis gelten sollen, die wir schon in der Aussagenlogik verwendet haben. Zusätzlich werden gleiche Quantoren zusammengefasst: Beispielsweise schreiben wir

$$\forall x,y\colon p(x,y) \quad \text{statt} \quad \forall x\colon (\forall y\colon p(x,y)).$$

Darüber hinaus legen wir fest, dass Quantoren stärker binden als die aussagenlogischen Junktoren. Damit können wir

$$\forall x\colon p(x) \wedge G \quad \text{statt} \quad \big(\forall x\colon p(x)\big) \wedge G$$

schreiben. Außerdem vereinbaren wir, dass wir zweistellige Prädikats- und Funktions-Zeichen auch in Infix-Notation angeben dürfen. Um eine eindeutige Lesbarkeit zu erhalten, müssen wir dann die Präzedenz der Funktions-Zeichen festlegen. Wir schreiben beispielsweise

$$\mathtt{n}_1 = \mathtt{n}_2 \quad \text{anstelle von} \quad =(\mathtt{n}_1, \mathtt{n}_2).$$

Die Formel $(\exists x\colon \leq(+(y,x),y))$ wird dann lesbarer als

$$\exists x\colon y + x \leq y$$

geschrieben. Außerdem finden Sie in der Literatur häufig Ausdrücke der Form $\forall x \in M : F$ oder $\exists x \in M : F$. Hierbei handelt es sich um Abkürzungen, die durch

$$\big(\forall x \in M : F\big) \overset{\text{def}}{\Longleftrightarrow} \forall x : \big(x \in M \rightarrow F\big), \quad \text{und} \quad \big(\exists x \in M : F\big) \overset{\text{def}}{\Longleftrightarrow} \exists x : \big(x \in M \wedge F\big).$$

definiert sind.

7.2 Semantik der Prädikatenlogik

Als nächstes legen wir die Bedeutung der Formeln fest. Dazu definieren wir den Begriff einer Σ-Struktur. Eine solche Struktur legt fest, wie die Funktions- und Prädikats-Zeichen der Signatur Σ zu interpretieren sind.

Definition 31 (Struktur) *Es sei eine Signatur*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \textit{arity} \rangle.$$

gegeben. Eine Σ-Struktur $\mathcal{S}$ ist ein Paar $\langle \mathcal{U}, \mathcal{J} \rangle$, so dass folgendes gilt:

1. *$\mathcal{U}$ ist eine nicht-leere Menge. Diese Menge nennen wir auch das Universum der Σ-Struktur. Dieses Universum enthält die Werte, die sich später bei der Auswertung der Terme ergeben werden.*
2. *$\mathcal{J}$ ist die Interpretation der Funktions– und Prädikats-Zeichen. Formal definieren wir $\mathcal{J}$ als eine Abbildung mit folgenden Eigenschaften:*
   (a) *Jedem Funktions-Zeichen $f \in \mathcal{F}$ mit $\textit{arity}(f) = m$ wird eine m-stellige Funktion*
   $$f^{\mathcal{J}} : \mathcal{U}^m \rightarrow \mathcal{U}$$
   zugeordnet, die m-Tupel des Universums $\mathcal{U}$ in das Universum $\mathcal{U}$ abbildet.
   (b) *Jedem Prädikats-Zeichen $p \in \mathcal{P}$ mit $\textit{arity}(p) = n$ wird eine Teilmenge*
   $$p^{\mathcal{J}} \subseteq \mathcal{U}^n$$
   zugeordnet. Die Idee ist, dass eine atomare Formel der Form $p(t_1, \cdots, t_n)$ genau dann als wahr interpretiert wird, wenn die Interpretation des Tupels $\langle t_1, \cdots, t_n \rangle$ ein Element der Menge $p^{\mathcal{J}}$ ist.
   (c) *Ist das Zeichen "$=$" ein Element der Menge der Prädikats-Zeichen $\mathcal{P}$, so gilt*
   $$=^{\mathcal{J}} \; = \; \big\{ \langle u, u \rangle \mid u \in \mathcal{U} \big\}.$$
   Eine Formel der Art $s = t$ wird also genau dann als wahr interpretiert, wenn die Interpretation des Terms s den selben Wert ergibt wie die Interpretation des Terms t. ◇

Beispiel: Die Signatur Σ_G der Gruppen-Theorie sei definiert als

$$\Sigma_G = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \textit{arity} \rangle \quad \text{mit}$$

1. $\mathcal{V} := \{x, y, z\}$

2. $\mathcal{F} := \{\mathrm{e}, *\}$

3. $\mathcal{P} := \{=\}$

4. $arity = \big\{\langle \mathrm{e}, 0\rangle, \langle *, 2\rangle, \langle =, 2\rangle\big\}$

Dann können wir eine Σ_G Struktur $\mathcal{Z} = \langle\{a, b\}, \mathcal{J}\rangle$ definieren, indem wir die Interpretation $\mathcal{J}$ wie folgt festlegen:

1. $\mathrm{e}^{\mathcal{J}} := a$,

2. $*^{\mathcal{J}} := \Big\{\big\langle\langle a, a\rangle, a\big\rangle, \big\langle\langle a, b\rangle, b\big\rangle, \big\langle\langle b, a\rangle, b\big\rangle, \big\langle\langle b, b\rangle, a\big\rangle\Big\}$,

3. $=^{\mathcal{J}} := \big\{\langle a, a\rangle, \langle b, b\rangle\big\}$.

   Beachten Sie, dass wir bei der Interpretation des Gleichheits-Zeichens keinen Spielraum haben!

$\diamond$

Falls wir Terme auswerten wollen, die Variablen enthalten, so müssen wir für diese Variablen irgendwelche Werte aus dem Universum einsetzen. Welche Werte wir einsetzen, kann durch eine Variablen-Belegung festgelegt werden. Diesen Begriff definieren wir nun.

Definition 32 (Variablen-Belegung) *Es sei eine Signatur*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity\rangle$$

gegeben. Weiter sei $\mathcal{S} = \langle \mathcal{U}, \mathcal{J}\rangle$ eine Σ-Struktur. Dann bezeichnen wir eine Abbildung

$$\mathcal{I} : \mathcal{V} \to \mathcal{U}$$

als eine $\mathcal{S}$-Variablen-Belegung.

Ist $\mathcal{I}$ eine $\mathcal{S}$-Variablen-Belegung, $x \in \mathcal{V}$ und $c \in \mathcal{U}$, so bezeichnet $\mathcal{I}[x/c]$ die Variablen-Belegung, die der Variablen x den Wert c zuordnet und die ansonsten mit $\mathcal{I}$ übereinstimmt:

$$\mathcal{I}[x/c](y) := \begin{cases} c & \text{falls } y = x; \\ \mathcal{I}(y) & \text{sonst.} \end{cases}$$

$\diamond$

Definition 33 (Semantik der Terme) *Ist $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ eine Σ-Struktur und $\mathcal{I}$ eine $\mathcal{S}$-Variablen-Belegung, so definieren wir für jeden Term t den Wert $\mathcal{S}(\mathcal{I}, t)$ durch Induktion über den Aufbau von t:*

1. *Für Variablen $x \in \mathcal{V}$ definieren wir:*

   $$\mathcal{S}(\mathcal{I}, x) := \mathcal{I}(x).$$

2. *Für Σ-Terme der Form $f(t_1, \cdots, t_n)$ definieren wir*

   $$\mathcal{S}\big(\mathcal{I}, f(t_1, \cdots, t_n)\big) := f^{\mathcal{J}}\big(\mathcal{S}(\mathcal{I}, t_1), \cdots, \mathcal{S}(\mathcal{I}, t_n)\big). \qquad \diamond$$

Beispiel: Mit der oben definieren Σ_G-Struktur $\mathcal{Z}$ definieren wir eine $\mathcal{Z}$-Variablen-Belegung $\mathcal{I}$ durch

$$\mathcal{I} := \big\{ \langle x, a \rangle, \langle y, b \rangle, \langle z, a \rangle \big\},$$

es gilt also

$$\mathcal{I}(\mathtt{x}) := a, \quad \mathcal{I}(\mathtt{y}) := b, \quad \text{und} \quad \mathcal{I}(\mathtt{z}) := a.$$

Dann gilt

$$\mathcal{Z}\big(\mathcal{I}, \mathtt{x} * \mathtt{y}\big) = b. \qquad \diamond$$

Definition 34 (Semantik der atomaren Σ-Formeln) *Ist $\mathcal{S}$ eine Σ-Struktur und $\mathcal{I}$ eine $\mathcal{S}$-Variablen-Belegung, so definieren wir für jede atomare Σ-Formel $p(t_1, \cdots, t_n)$ den Wert $\mathcal{S}\big(\mathcal{I}, p(t_1, \cdots, t_n)\big)$ wie folgt:*

$$\mathcal{S}\big(\mathcal{I}, p(t_1, \cdots, t_n)\big) := \Big(\big\langle \mathcal{S}(\mathcal{I}, t_1), \cdots, \mathcal{S}(\mathcal{I}, t_n) \big\rangle \in p^{\mathcal{J}}\Big). \qquad \diamond$$

Beispiel: In Fortführung des obigen Beispiels gilt:

$$\mathcal{Z}(\mathcal{I}, x * y = y * x) = \mathtt{True}. \qquad \diamond$$

Um die Semantik beliebiger Σ-Formeln definieren zu können, nehmen wir an, dass wir, genau wie in der Aussagenlogik, die folgenden Funktionen zur Verfügung haben:

1. $\ominus : \mathbb{B} \to \mathbb{B}$,
2. $\circledvee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$,
3. $\circledwedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$,
4. $\ominus\!\!\!\rightarrow : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$,
5. $\leftrightarrow\!\!\!\!\!\bigcirc : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$.

Die Semantik dieser Funktionen hatten wir durch die Tabelle in Abbildung 6.1 auf Seite 78 gegeben.

Definition 35 (Semantik der Σ-Formeln) *Ist $\mathcal{S}$ eine Σ-Struktur und $\mathcal{I}$ eine $\mathcal{S}$-Variablen-Belegung, so definieren wir für jede Σ-Formel F den Wert $\mathcal{S}(\mathcal{I}, F)$ durch Induktion über den Aufbau von F:*

1. $\mathcal{S}(\mathcal{I}, \top) := \mathtt{True}$ *und* $\mathcal{S}(\mathcal{I}, \bot) := \mathtt{false}$.
2. $\mathcal{S}(\mathcal{I}, \neg F) := \ominus\big(\mathcal{S}(\mathcal{I}, F)\big)$.

3. $\mathcal{S}(\mathcal{I}, F \wedge G) := \textcircled{\wedge}(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.

4. $\mathcal{S}(\mathcal{I}, F \vee G) := \textcircled{\vee}(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.

5. $\mathcal{S}(\mathcal{I}, F \rightarrow G) := \ominus\!\!\rightarrow(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.

6. $\mathcal{S}(\mathcal{I}, F \leftrightarrow G) := \textcircled{\leftrightarrow}(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.

7. $\mathcal{S}(\mathcal{I}, \forall x\colon F) := \begin{cases} \texttt{True} & \textit{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \texttt{True} \quad \textit{für alle } c \in \mathcal{U} \textit{ gilt;} \\ \texttt{false} & \textit{sonst.} \end{cases}$

8. $\mathcal{S}(\mathcal{I}, \exists x\colon F) := \begin{cases} \texttt{True} & \textit{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \texttt{True} \quad \textit{für ein } c \in \mathcal{U} \textit{ gilt;} \\ \texttt{false} & \textit{sonst.} \end{cases}$ ◇

Beispiel: In Fortführung des obigen Beispiels gilt

$$\mathcal{S}(\mathcal{I}, \forall \mathtt{x} : \mathtt{e} * x = x) = \texttt{True}.$$ ◇

Definition 36 (Allgemeingültig) *Ist F eine Σ-Formel, so dass für jede Σ-Struktur $\mathcal{S}$ und für jede $\mathcal{S}$-Variablen-Belegung $\mathcal{I}$*

$$\mathcal{S}(\mathcal{I}, F) = \texttt{True}$$

gilt, so bezeichnen wir F als allgemeingültig. In diesem Fall schreiben wir

$$\models F.$$ ◇

Ist F eine Formel für die $FV(F) = \{\}$ ist, dann hängt der Wert $\mathcal{S}(\mathcal{I}, F)$ offenbar gar nicht von der Interpretation $\mathcal{I}$ ab. Solche Formeln bezeichnen wir auch als geschlossene Formeln. In diesem Fall schreiben wir kürzer $\mathcal{S}(F)$ an Stelle von $\mathcal{S}(\mathcal{I}, F)$. Gilt dann zusätzlich $\mathcal{S}(F) = \texttt{True}$, so sagen wir auch, dass $\mathcal{S}$ ein Modell von F ist. Wir schreiben dann

$$\mathcal{S} \models F.$$

Die Definition der Begriffe "erfüllbar" und "äquivalent" lassen sich nun aus der Aussagenlogik übertragen. Um unnötigen Ballast in den Definitionen zu vermeiden, nehmen wir im Folgenden immer eine feste Signatur Σ als gegeben an. Dadurch können wir in den folgenden Definitionen von Termen, Formeln, Strukturen, etc. sprechen und meinen damit Σ-Terme, Σ-Formeln und Σ-Strukturen.

Definition 37 (Äquivalent) *Zwei Formeln F und G, in denen die Variablen $x_1, \cdots, x_n$ frei auftreten, heißen äquivalent g.d.w.*

$$\models \forall x_1 : \cdots \forall x_n : (F \leftrightarrow G)$$

gilt. Falls in F und G keine Variablen frei auftreten, dann ist F genau dann äquivalent zu G, wenn

$$\models F \leftrightarrow G$$

gilt. ◇

Bemerkung: Alle aussagenlogischen Äquivalenzen sind auch prädikatenlogische Äquivalenzen. ◇

Definition 38 (Erfüllbar) *Eine Menge $M \subseteq \mathbb{F}_\Sigma$ ist genau dann erfüllbar, wenn es eine Struktur $\mathcal{S}$ und eine Variablen-Belegung $\mathcal{I}$ gibt, so dass*

$$\mathcal{S}(\mathcal{I}, F) = \texttt{True} \quad \textit{für alle } F \in M$$

gilt. Andernfalls heißt M unerfüllbar oder auch widersprüchlich. Wir schreiben dafür auch

$$M \models \bot \qquad\diamond$$

Unser Ziel ist es, ein Verfahren anzugeben, mit dem wir in der Lage sind zu überprüfen, ob eine Menge M von Formeln widersprüchlich ist, ob also $M \models \bot$ gilt. Es zeigt sich, dass dies im Allgemeinen nicht möglich ist, die Frage, ob $M \models \bot$ gilt, ist unentscheidbar. Ein Beweis dieser Tatsache geht allerdings über den Rahmen dieser Vorlesung heraus. Dem gegenüber ist es möglich, ähnlich wie in der Aussagenlogik einen Kalkül $\vdash$ anzugeben, so dass gilt:

$$M \vdash \bot \quad \text{g.d.w.} \quad M \models \bot.$$

Ein solcher Kalkül kann dann zur Implementierung eines Semi-Entscheidungs-Verfahrens benutzt werden: Um zu überprüfen, ob $M \models \bot$ gilt, versuchen wir, aus der Menge M die Formel $\bot$ herzuleiten. Falls wir dabei systematisch vorgehen, indem wir alle möglichen Beweise durchprobieren, so werden wir, falls tatsächlich $M \models \bot$ gilt, auch irgendwann einen Beweis finden, der $M \vdash \bot$ zeigt. Wenn allerdings der Fall

$$M \not\models \bot$$

vorliegt, so werden wir dies im Allgemeinen nicht feststellen können, denn die Menge aller Beweise ist unendlich und wir können nie alle Beweise ausprobieren. Wir können lediglich sicherstellen, dass wir jeden Beweis irgendwann versuchen. Wenn es aber keinen Beweis gibt, so können wir das nie sicher sagen, denn zu jedem festen Zeitpunkt haben wir ja immer nur einen Teil der in Frage kommenden Beweise ausprobiert.

Die Situation ist ähnlich der, wie bei der Überprüfung bestimmter zahlentheoretischer Fragen. Wir betrachten dazu ein konkretes Beispiel: Eine Zahl n heißt eine perfekte Zahl, wenn die Summe aller echten Teiler von n wieder die Zahl n ergibt. Beispielsweise ist die Zahl 6 perfekt, denn die Menge der echten Teiler von 6 ist $\{1,2,3\}$ und es gilt

$$1 + 2 + 3 = 6.$$

Bisher sind alle bekannten perfekten Zahlen durch 2 teilbar. Die Frage, ob es auch ungerade Zahlen gibt, die perfekt sind, ist ein offenes mathematisches Problem. Um dieses Problem zu lösen, könnten wir eine Programm schreiben, dass der Reihe nach für alle ungerade Zahlen überprüft, ob die Zahl perfekt ist. Abbildung 7.1 auf Seite 140 zeigt ein solches Programm. Wenn es eine ungerade perfekte Zahl gibt, dann wird dieses Programm diese Zahl auch irgendwann finden. Wenn es aber keine ungerade perfekte Zahl gibt, dann wird das Programm bis zum St. Nimmerleinstag rechnen und wir werden nie mit Sicherheit wissen, dass es keine ungeraden perfekten Zahlen gibt.

7.3 Implementierung prädikatenlogischer Strukturen in *Python*

Der im letzten Abschnitt präsentierte Begriff einer prädikatenlogischen Struktur erscheint zunächst sehr abstrakt. Wir wollen in diesem Abschnitt zeigen, dass sich dieser Begriff in einfacher Weise in *Python* implementieren lässt. Dadurch gelingt es, diesen Begriff zu veranschaulichen. Als konkretes Beispiel wollen wir Strukturen zu Gruppen-Theorie betrachten. Wir gehen dazu in vier Schritten vor:

1. Zunächst definieren wir mathematisch, was wir unter einer *Gruppe* verstehen.
2. Anschließend diskutieren wir, wie wir die Formeln der Gruppen-Theorie in *Python* darstellen.
3. Dann definieren wir eine Struktur, in der die Formeln der Gruppen-Theorie gelten.

```
def perfect(n):
    return sum({ x for x in range(1, n) if n % x == 0 }) == n

def findOddPerfect():
    n = 1
    while True:
        if perfect(n):
            return n
        n += 2

findOddPerfect()
```

Figure 7.1: Suche nach einer ungeraden perfekten Zahl.

4. Schließlich zeigen wir, wie wir prädikaten-logische Formeln in *Python* auswerten können.

7.3.1 Gruppen-Theorie

In der Mathematik wird eine Gruppe $\mathcal{G}$ als ein Tripel der Form

$$\mathcal{G} = \langle G, \mathrm{e}, * \rangle$$

definiert. Dabei gilt:

1. G ist eine Menge,
2. e ist ein Element der Menge G und
3. $* : G \times G \to G$ ist eine binäre Funktion auf G.
4. Zusätzlich müssen die folgenden Axiome gelten:
   (a) $\forall x : \mathrm{e} * x = x$,
   (b) $\forall x : \exists y : y * x = \mathrm{e}$,
   (c) $\forall x : \forall y : \forall z : (x * y) * z = x * (y * z)$.
   (d) Die Gruppe $\mathcal{G}$ ist eine *kommutative* Gruppe genau dann, wenn zusätzlich das folgende Axiom gilt:
   $\forall x : \forall y : x * y = y * x$.

7.3.2 Darstellung der Formeln in *Python*

Im letzten Abschnitt haben wir die Signatur Σ_G der Gruppen-Theorie wie folgt definiert:

$$\Sigma_G = \big\langle \{x, y, z\},\ \{\mathrm{e}, *\},\ \{=\},\ \{\langle \mathrm{e}, 0\rangle, \langle *, 2\rangle, \langle =, 2\rangle\}\big\rangle.$$

Hierbei ist also "e" ein 0-stelliges Funktions-Zeichen, "$*$" ist eine 2-stelliges Funktions-Zeichen und "$=$" ist ein 2-stelliges Prädikats-Zeichen. Wir werden für prädikaten-logische Formeln einen Parser verwenden, der keine binären Operatoren wie "$*$" oder "$=$" unterstützt. Bei diesem Parser können Terme nur in der Form

$f(t_1, \cdots, t_n)$

angegeben werden, wobei f eine Funktions-Zeichen ist und $t_1, \cdots, t_n$ Terme sind. Analog werden atomare Formeln durch Ausdrücke der Form

$p(t_1, \cdots, t_n)$

dargestellt, wobei p eine Prädikats-Zeichen ist. Variablen werden von den Funktions- und Prädikats-Zeichen dadurch unterschieden, dass Variablen mit einem kleinen Buchstaben beginnen, während Funktions- und Prädikats-Zeichen mit einem großen Buchstaben beginnen. Um die Formeln der Gruppentheorie darstellen zu können, vereinbaren wir daher Folgendes:

1. Das neutrale Element e schreiben wir als `E()`.
2. Für den Operator $*$ verwenden wir das zweistellige Funktions-Zeichen `Multiply`. Damit wird der Ausdruck $x * y$ also als `Multiply`(x, y) geschrieben.
3. Das Gleichheits-Zeichen "=" repräsentieren wir durch das zweistellige Prädikats-Zeichen `Equals`. Damit schreibt sich die Formel $x = y$ dann als `Equals`(x, y).

Abbildung 7.2 zeigt die Formeln der Gruppen-Theorie als Strings.

```
G1 = '∀x:Equals(Multiply(E(),x),x)'
G2 = '∀x:∃y:Equals(Multiply(x,y),E())'
G3 = '∀x:∀y:∀z:Equals(Multiply(Multiply(x,y),z), Multiply(x,Multiply(y,z)))'
G4 = '∀x:∀y:Equals(Multiply(x,y), Multiply(y,x))'
```

Figure 7.2: Die Formeln der kommutativen Gruppentheorie als Strings

Wir können die Formeln mit der in Abbildung 7.3 gezeigten Funktion `parse`(s) in geschachtelte Tupel überführen. Das Ergebnis dieser Transformation ist in Abbildung 7.4 zu sehen.

```
import folParser as fp

def parse(s):
    "Parse string s as fol formula."
    p = fp.LogicParser(s)
    return p.parse()

F1 = parse(G1)
F2 = parse(G2)
F3 = parse(G3)
F4 = parse(G4)
```

Figure 7.3: Die Funktion `parse`

```
F1 = ('∀', 'x', ('Equals', ('Multiply', ('E',), 'x'), 'x'))
F2 = ('∀', 'x', ('∃', 'y', ('Equals', ('Multiply', 'x', 'y'), ('E',))))
F3 = ('∀', 'x', ('∀', 'y', ('∀', 'z',
          ('Equals', ('Multiply', ('Multiply', 'x', 'y'), 'z'),
                     ('Multiply', 'x', ('Multiply', 'y', 'z'))
          )
     )))
F4 = ('∀', 'x', ('∀', 'y',
          ('Equals', ('Multiply', 'x', 'y'),
                     ('Multiply', 'y', 'x')
          )
     ))
```

Figure 7.4: Die Axiome einer kommutativen Gruppe als geschachtelte Tupel

7.3.3 Darstellung prädikaten-logischer Strukturen in *Python*

Wir hatten bei der Definition der Semantik Prädikaten-Logik in Abschnitt 7.2 bereits eine Struktur $\mathcal{S}$ angegeben, deren Universum aus der Menge $\{a, b\}$ besteht. In *Python* können wir diese Struktur durch den in Abbildung 7.5 auf Seite 142 gezeigten Code implementieren.

```
1  a = "a"
2  b = "b"
3  U = { a, b }
4  NeutralElement = { (): a }
5  Product        = { (a, a): a,  (a, b): b,  (b, a): b,  (b, b): a }
6  Identity       = { (x, x) for x in U }
7  J = { "E": NeutralElement, "Multiply": Product, "Equals": Identity }
8  S = (U, J)
9  I = { "x": a, "y": b, "z": a }
```

Figure 7.5: Implementierung einer Struktur zur Gruppen-Theorie

1. Zur Abkürzung haben wir in den Zeile 1 und 2 die Variablen `a` und `b` als die Strings `"a"` und `"b"` definiert. Dadurch können wir weiter unten die Interpretation des Funktions-Zeichens "`Multiply`" einfacher angeben.

2. Das in Zeile 3 definierte Universum U besteht aus den beiden Strings `"a"` und `"b"`.

3. In Zeile 4 definieren wir die Interpretation des nullstelligen Funktions-Zeichens `E` als das *Python*-Dictionary, das dem leeren Tupel das Objekt `a` zuordnet.

4. In Zeile 5 definieren wir eine Funktion `Product` als *Python*-Dictionary. Für die so definierte Funktion gilt

$$\texttt{Product}(\texttt{"a"},\texttt{"a"}) = \texttt{"a"}, \quad \texttt{Product}(\texttt{"a"},\texttt{"b"}) = \texttt{"b"},$$
$$\texttt{Product}(\texttt{"b"},\texttt{"a"}) = \texttt{"b"}, \quad \texttt{Product}(\texttt{"b"},\texttt{"b"}) = \texttt{"a"}.$$

Diese Funktion verwenden wir später als die Interpretation $\texttt{Multiply}^{\mathcal{J}}$ des Funktions-Zeichens "`Multiply`".

5. In Zeile 6 haben wir die Interpretation $\texttt{Equals}^{\mathcal{J}}$ des Prädikats-Zeichens "`Equals`" als Menge aller Paare der Form (x, x) dargestellt, wobei x ein beliebiges Element des Universums ist.

6. In Zeile 7 fassen wir die Interpretationen der Funktions-Zeichen "`E`" und "`Multiply`" und des Prädikats-Zeichens "`Equals`" zu dem Dictionary `J` zusammen, so dass für ein Funktions- oder Prädikats-Zeichen f die Interpretation $f^{\mathcal{J}}$ durch den Wert $\texttt{J}[f]$ gegeben ist.

7. Die Interpretation `J` wird dann in Zeile 8 mit dem Universum `U` zu der Struktur `S` zusammengefasst, die in *Python* einfach als Paar dargestellt wird.

8. Schließlich zeigt Zeile 9, dass eine Variablen-Belegung ebenfalls als Dictionary dargestellt werden kann. Die Schlüssel sind die Variablen, die Werte sind dann die Objekte aus dem Universum, auf welche die Variablen abgebildet werden.

```
1   def evalTerm(t, S, I):
2       if isinstance(t, str):  # t is a variable
3           return I[t]
4       J        = S[1]   # dictionary of interpretations
5       f        = t[0]   # function symbol
6       fJ       = J[f]   # interpretation of function symbol
7       argTuple = t[1:]
8       argVals  = evalTermTuple(argTuple, S, I)
9       return fJ[argVals]
10
11  def evalTermTuple(Ts, S, I):
12      return tuple(evalTerm(t, S, I) for t in Ts)
```

Figure 7.6: Auswertung von Termen

Als nächstes überlegen wir uns, wie wir prädikatenlogische Terme in einer solchen Struktur auswerten können. Abbildung 7.6 zeigt die Implementierung der Prozedur $\texttt{evalTerm}(t, \mathcal{S}, \mathcal{I})$, der als Argumente ein prädikatenlogischer Term t, eine prädikatenlogische Struktur $\mathcal{S}$ und eine Variablen-Belegung $\mathcal{I}$ übergeben werden. Der Term t wird dabei in *Python* als geschachteltes Tupel dargestellt.

1. In Zeile 2 überprüfen wir, ob der Term t eine Variable ist. Dies ist daran zu erkennen, dass Variablen als Strings dargestellt werden, während alle anderen Terme Tupel sind. Falls t eine Variable ist, dann geben wir den Wert zurück, der in der Variablen-Belegung $\mathcal{I}$ für diese Variable gespeichert ist.

2. Sonst extrahieren wir in Zeile 4 das Dictionary $\mathcal{J}$, das die Interpretationen der Funktions- und Prädikats-Zeichen enthält, aus der Struktur $\mathcal{S}$.

3. Das Funktions-Zeichen f des Terms t ist die erste Komponente des Tupels t.

4. Die Interpretation $f^{\mathcal{J}}$ dieses Funktions-Zeichens schlagen wir in Zeile 6 in dem Dictionary $\mathcal{J}$ nach.

5. Die Argumente des Funktions-Zeichens f sind die restlichen Komponenten des Tupels t.
6. Das Tupel, das aus diesen Argumenten besteht, wird in Zeile 8 rekursiv ausgewertet. Als Ergebnis erhalten wir dabei ein Tupel von Werten.
7. Dieses Tupel dient dann in Zeile 9 als Argument für das Dictionary $f^{\mathcal{J}}$. Der in diesem Dictionary für die Argumente abgelegte Wert ist das Ergebnis der Auswertung des Terms t.

```
1  def evalAtomic(a, S, I):
2      J        = S[1]  # dictionary of interpretations
3      p        = a[0]  # predicate symbol
4      pJ       = J[p]  # interpretation of predicate symbol
5      argTuple = a[1:]
6      argVals  = evalTermTuple(argTuple, S, I)
7      return argVals in pJ
```

Figure 7.7: Auswertung atomarer Formeln

Abbildung 7.7 zeigt die Auswertung einer atomaren Formel. Eine atomare Formel a ist in Python als Tupel der Form

$$a = (p, t_1, \cdots, t_n).$$

dargestellt. Das Prädikats-Zeichen ergibt sich daher als $a[0]$, während das Tupel der Argumente durch den Ausdruck $a[1:]$ gegeben ist. Um zu überprüfen, ob die atomare Formel a wahr ist, müssen wir überprüfen, ob

$$\big(\texttt{evalTerm}(t_1, \mathcal{S}, \mathcal{I}), \cdots, \texttt{evalTerm}(t_n, \mathcal{S}, \mathcal{I})\big) \in p^{\mathcal{J}}$$

gilt. Dieser Test wird in Zeile 7 von Abbildung durchgeführt. Der Rest der Implementierung der Funktion `evalAtomic` ist analog zur Implementierung der Funktion `evalTerm`.

Abbildung 7.8 auf Seite 145 zeigt die Implementierung der Funktion $\texttt{evalFormula}(F, \mathcal{S}, \mathcal{I})$, die als Argumente eine prädikatenlogische Formel F, eine prädikatenlogische Struktur $\mathcal{S}$ und eine Variablen-Belegung $\mathcal{I}$ erhält und die als Ergebnis den Wert $\mathcal{S}(\mathcal{I}, F)$ berechnet. Die Auswertung der Formel F erfolgt dabei analog zu der in Abbildung 6.1 auf Seite 82 gezeigten Auswertung aussagenlogischer Formeln. Neu ist hier nur die Behandlung der Quantoren. In den Zeilen 10, 11 und 12 behandeln wir die Auswertung allquantifizierter Formeln. Ist F eine Formel der Form $\forall x : G$, so wird die Formel F durch das Tupel

$$F = (\text{'}\forall\text{'}, x, G)$$

dargestellt. Die Auswertung von $\forall x \colon G$ geschieht nach der Formel

$$\mathcal{S}(\mathcal{I}, \forall x \colon G) := \begin{cases} \texttt{True} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], G) = \texttt{True} \quad \text{für alle } c \in \mathcal{U} \text{ gilt;} \\ \texttt{false} & \text{sonst.} \end{cases}$$

Um die Auswertung implementieren zu können, verwenden wir die Prozedur `modify()`, welche die Variablen-Belegung $\mathcal{I}$ an der Stelle x zu c abändert, es gilt also

$$\texttt{modify}(\mathcal{I}, x, c) = \mathcal{I}[x/c].$$

Die Implementierung dieser Prozedur ist in Abbildung 7.9 auf Seite 145 gezeigt. Bei der Auswertung eines All-Quantors können wir ausnutzen, dass die Sprache *Python* den Quantor "∀" durch die

```
def evalFormula(F, S, I):
    U = S[0]  # universe
    if F[0] == '⊤': return True
    if F[0] == '⊥': return False
    if F[0] == '¬': return not evalFormula(F[1], S, I)
    if F[0] == '∧': return evalFormula(F[1], S, I) and evalFormula(F[2], S, I)
    if F[0] == '∨': return evalFormula(F[1], S, I) or evalFormula(F[2], S, I)
    if F[0] == '→': return not evalFormula(F[1], S, I) or evalFormula(F[2], S, I)
    if F[0] == '↔': return evalFormula(F[1], S, I) == evalFormula(F[2], S, I)
    if F[0] == '∀':
        x, G = F[1:]
        return all({ evalFormula(G, S, modify(I, x, c)) for c in U} )
    if F[0] == '∃':
        x, G = F[1:]
        return any({ evalFormula(G, S, modify(I, x, c)) for c in U} )
    return evalAtomic(F, S, I)
```

Figure 7.8:

Funktion `all` unterstützt. Wir können also direkt testen, ob die Formel für alle möglichen Werte c, die wir für die Variable x einsetzen können, richtig ist. Für eine Menge S von Wahrheitswerten ist der Ausdruck

$\texttt{all}(S)$

genau dann wahr, wenn alle Elemente von S den Wert `True` haben. Die Auswertung eines Existenz-Quantors ist analog zur Auswertung eines All-Quantors. Der einzige Unterschied besteht darin, dass wir statt der Funktion `all` die Funktion `any` verwenden. Der Ausdruck

$\texttt{any}(S)$

ist für eine Menge von Wahrheitswerten S genau dann wahr, wenn es wenigstens ein Element in der Menge S gibt, dass den Wert `True` hat.

Bei der Implementierung der Prozedur $\texttt{modify}(I, x, c)$, die als Ergebnis die Variablen-Belegung $\mathcal{I}[x/c]$ berechnet, nutzen wir aus, dass wir bei einer Funktion, die als Dictionary gespeichert ist, den Wert, der für ein Argument x eingetragen ist, durch eine Zuweisung der Form

$\mathcal{I}[x] = c$

abändern können.

```
def modify(I, x, c):
    I[x] = c
    return I
```

Figure 7.9: Die Implementierung der Funktion `modify`.

Mit dem in Abbildung 7.10 gezeigten Skript können wir nun überprüfen, ob die in Abbildung 7.10 auf Seite 146 definierte Struktur eine Gruppe ist. Wir erhalten die in Abbildung 7.11 gezeigte

Ausgabe uns können daher folgern, dass diese Struktur in der Tat eine kommutative Gruppe ist.

```
f"evalFormula({G1}, S, I) = {evalFormula(F1, S, I)}"
f"evalFormula({G2}, S, I) = {evalFormula(F2, S, I)}"
f"evalFormula({G3}, S, I) = {evalFormula(F3, S, I)}"
f"evalFormula({G4}, S, I) = {evalFormula(F4, S, I)}"
```

Figure 7.10: Überprüfung, ob die in Abbildung 7.5 definierte Struktur eine Gruppe ist

```
evalFormula(∀x:Equals(Multiply(E(),x),x), S, I) = True
evalFormula(∀x:∃y:Equals(Multiply(x,y),E()), S, I) = True
evalFormula(∀x:∀y:∀z:Equals(Multiply(Multiply(x,y),z), Multiply(x,Multiply(y,z))), S, I)
= True
evalFormula(∀x:∀y:Equals(Multiply(x,y), Multiply(y,x)), S, I) = True
```

Figure 7.11: Ausgabe des in Abbildung 7.10 gezeigten Skripts

Bemerkung: Das oben vorgestellte Programm finden sie als Jupyter Notebook auf GitHub unter der Adresse:

`https://github.com/karlstroetmann/Logic/blob/master/FOL-Evaluation.ipynb`

Mit diesem Programm können wir überprüfen, ob eine prädikatenlogische Formel in einer vorgegebenen endlichen Struktur erfüllt ist. Wir können damit allerdings nicht überprüfen, ob eine Formel allgemeingültig ist, denn einerseits können wir das Programm nicht anwenden, wenn die Strukturen ein unendliches Universum haben, andererseits ist selbst die Zahl der verschiedenen endlichen Stukturen, die wir ausprobieren müssten, unendlich groß. ◇

Aufgabe 11:

1. Zeigen Sie, dass die Formel

   $$\forall x : \exists y : p(x,y) \rightarrow \exists y : \forall x : p(x,y)$$

   nicht allgemeingültig ist, indem Sie in *Python* eine geeignete prädikatenlogische Struktur $\mathcal{S}$ implementieren, in der diese Formel falsch ist.

2. Überlegen Sie, wie viele verschiedene Strukturen mit n Elementen es für die obige Formel gibt.

3. Geben Sie eine erfüllbare prädikatenlogische Formel F an, die in einer prädikatenlogischen Struktur $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ immer falsch ist, wenn das Universum $\mathcal{U}$ endlich ist.

   Hinweis: Es sei $f : U \rightarrow U$ eine Funktion. Überlegen Sie, wie die Aussagen "*f ist injektiv*" und "*f ist surjektiv*" zusammen hängen. ◇

7.4 Constraint Programing

It is time to see an application of first order logic. One of these applications is constraint programming. Constraint programming is an example of the declarative programming paradigm. The idea is that in order to solve a given problem, this problem is specified and then this specification is given as input to a problem solver which will then compute a solution to the problem. Hence, the task of the programmer is much easier than it normally is: Instead of implementing a program that solves a given problem, the programmer only has to specify the problem precisely, she does not have to specify an algorithm to find the solution. Usually, the specification of a problem is much easier than the specification of an algorithm to solve the problem. This approach works well for those problems that can be specified using first order logic. The remainder of this section is structured as follows:

1. We first define constraint satisfaction problems.

   As an example, we show how the eight queens puzzle can be formulated as a constraint satisfaction problem.

2. We discuss a simple constraint solver that is based on backtracking.

7.4.1 Constraint Satisfaction Problems

Conceptually, a constraint satisfaction problem is given by a set of first order logic formulas that contain a number of free variables. Furthermore, a structure $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ consisting of a universe $\mathcal{U}$ and the interpretation $\mathcal{J}$ of the function and predicate symbols used in these formulas is assumed to be understood from the context of the problem. The goal is to find a variable assignment such that the given formulas are evaluated as true.

Definition 39 (CSP)
Formally, a constraint satisfaction problem (abbreviated as CSP*) is defined as a triple*

$$\mathcal{P} := \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle$$

where

1. `Vars` *is a set of strings which serve as variables,*

2. `Values` *is a set of values that can be assigned to the variables in* `Vars`*.*

   This set of values is assumed to be identical to the universe of the first order structure $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ *that is given implicitly, i.e. we have*

   $$\texttt{Values} = \mathcal{U}.$$

3. `Constraints` *is a set of formulas from first order logic. Each of these formulas is called a constraint of* $\mathcal{P}$*.* ◇

Given a CSP

$$\mathcal{P} = \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle,$$

a variable assignment for $\mathcal{P}$ is a function

$$\mathcal{I} : \texttt{Vars} \to \texttt{Values}.$$

A variable assignment $\mathcal{I}$ is a solution of the CSP $\mathcal{P}$ if, given the assignment $\mathcal{I}$, all constraints of $\mathcal{P}$ are satisfied, i.e. we have

$$\mathcal{S}(\mathcal{I}, f) = \texttt{True} \quad \text{for all } f \in \texttt{Constraints}.$$

Finally, a partial variable assignment $\mathcal{B}$ for $\mathcal{P}$ is a function

$$\mathcal{B} : \texttt{Vars} \to \texttt{Values} \cup \{\Omega\} \quad \text{where } \Omega \text{ denotes the undefined value.}$$

Hence, a partial variable assignment does not assign values to all variables. Instead, it assigns values only to a subset of the set `Vars`. The domain $\texttt{dom}(\mathcal{B})$ of a partial variable assignment $\mathcal{B}$ is the set of those variables that are assigned a value different from Ω, i.e. we define

$$\texttt{dom}(\mathcal{B}) := \{x \in \texttt{Vars} \mid \mathcal{B}(x) \neq \Omega\}.$$

We proceed to illustrate the definitions given so far by presenting two examples.

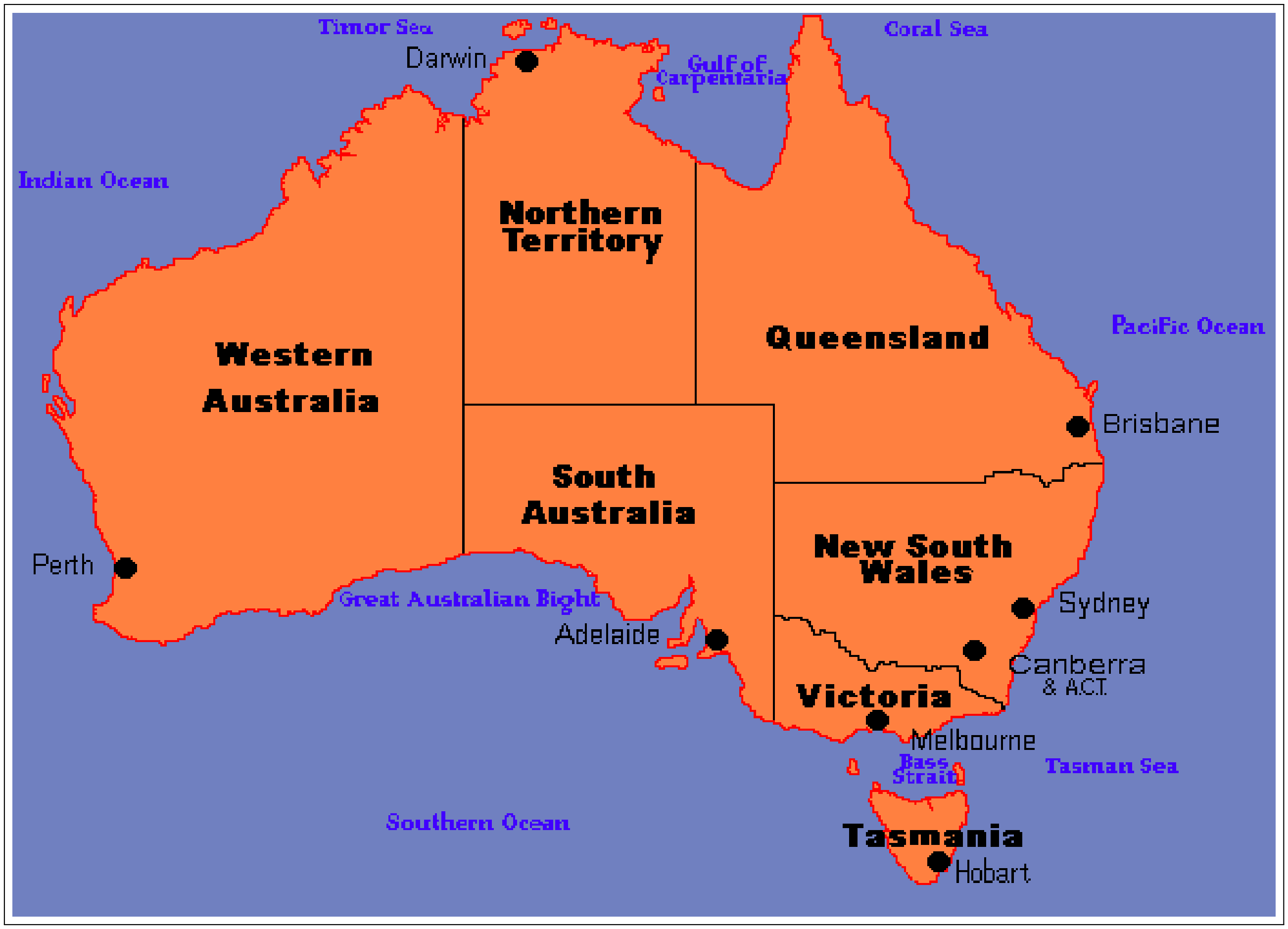

Figure 7.12: A map of Australia.

7.4.2 Example: Map Colouring

In map colouring a map showing different state borders is given and the task is to colour the different states such that no two states that have a common border share the same colour. Figure 7.12 on page 148 shows a map of Australia. There are seven different states in Australia:

1. Western Australia, abbreviated as WA,
2. Northern Territory, abbreviated as NT,
3. South Australia, abbreviated as SA,

4. Queensland, abbreviated as Q,
5. New South Wales, abbreviated as NSW,
6. Victoria, abbreviated as V, and
7. Tasmania, abbreviated as T.

Figure 7.12 would certainly look better if different states had been coloured with different colours. For the purpose of this example let us assume that we have only three colours available. The question then is whether it is possible to colour the different states in a way that no two neighbouring states share the same colour. This problem can be formalized as a constraint satisfaction problem. To this end we define:

1. $\texttt{Vars} := \{\mathrm{WA}, \mathrm{NT}, \mathrm{SA}, \mathrm{Q}, \mathrm{NSW}, \mathrm{V}, \mathrm{T}\}$,
2. $\texttt{Values} := \{\texttt{red}, \texttt{green}, \texttt{blue}\}$,
3. $\texttt{Constraints} :=$
   $\{\mathrm{WA} \neq \mathrm{NT}, \mathrm{WA} \neq \mathrm{SA}, \mathrm{NT} \neq \mathrm{SA}, \mathrm{NT} \neq \mathrm{Q}, \mathrm{SA} \neq \mathrm{Q}, \mathrm{SA} \neq \mathrm{NSW}, \mathrm{SA} \neq \mathrm{V}, \mathrm{Q} \neq \mathrm{NSW}, \texttt{NSW} \neq \texttt{V}, \mathrm{V} \neq \mathrm{T}\}$

Then $\mathcal{P} := \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle$ is a constraint satisfaction problem. If we define the assignment $\mathcal{I}$ such that

1. $\mathcal{I}(\mathrm{WA}) = \texttt{blue}$,
2. $\mathcal{I}(\mathrm{NT}) = \texttt{red}$,
3. $\mathcal{I}(\mathrm{SA}) = \texttt{green}$,
4. $\mathcal{I}(\mathrm{Q}) = \texttt{blue}$,
5. $\mathcal{I}(\mathrm{NSW}) = \texttt{red}$,
6. $\mathcal{I}(\mathrm{V}) = \texttt{blue}$,
7. $\mathcal{I}(\mathrm{T}) = \texttt{red}$,

then you can check that the assignment $\mathcal{I}$ is indeed a solution to the constraint satisfaction problem $\mathcal{P}$.

7.4.3 Example: The Eight Queens Puzzle

The eight queens problem asks to put 8 queens onto a chessboard such that no queen can attack another queen. In chess, a queen can attack all pieces that are either in the same row, the same column, or the same diagonal. If we want to put 8 queens on a chessboard such that no two queens can attack each other, we have to put exactly one queen in every row: If we would put more than one queen in a row, the queens in that row can attack each other. If we would leave a row empty, then, given that the other rows contain at most one queen, there would be less than 8 queens on the board. Therefore, in order to model the eight queens problem as a constraint satisfaction problem, we will use the following set of variables:

$\texttt{Vars} := \{\texttt{Q}_1, \texttt{Q}_2, \texttt{Q}_3, \texttt{Q}_4, \texttt{Q}_5, \texttt{Q}_6, \texttt{Q}_7, \texttt{Q}_8\}$,

where for $i \in \{1, \cdots, 8\}$ the variable $\texttt{Q}_i$ specifies the column of the queen that is placed in row i. As the columns run from one to eight, we define the set `Values` as

$\texttt{Values} := \{1, 2, 3, 4, 5, 6, 7, 8\}$.

Next, let us define the constraints. There are two different types of constraints.

1. We have constraints that express that no two queens positioned in different rows share the same column. To capture these constraints, we define

   $\texttt{SameRow} := \bigl\{\texttt{Q}_i \neq \texttt{Q}_j \bigm| i \in \{1, \cdots, 8\} \wedge j \in \{1, \cdots, 8\} \wedge j < i\bigr\}$.

   Here the condition $i < j$ ensures that, for example, we have the constraint $\texttt{Q}_2 \neq \texttt{Q}_1$ but not the constraint $\texttt{Q}_1 \neq \texttt{Q}_2$, as the latter constraint would be redundant if the former constraint has already been established.

2. We have constraints that express that no two queens positioned in different rows share the same diagonal. The queens in row i and row j share the same diagonal iff the equation

   $|i - j| = |\texttt{Q}_i - \texttt{Q}_j|$

   holds. The expression $|i - j|$ is the absolute value of the difference of the rows of the queens in row i and row j, while the expression $|\texttt{Q}_i - \texttt{Q}_j|$ is the absolute value of the difference of the columns of these queens. To capture these constraints, we define

   $\texttt{SameDiagonal} := \bigl\{|i - j| \neq |\texttt{Q}_i - \texttt{Q}_j| \bigm| i \in \{1, \cdots, 8\} \wedge j \in \{1, \cdots, 8\} \wedge j < i\bigr\}$.

Then, the set of constraints is defined as

$\texttt{Constraints} := \texttt{SameRow} \cup \texttt{SameDiagonal}$

and the eight queens problem can be stated as the constraint satisfaction problem

$\mathcal{P} := \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle$.

If we define the assignment $\mathcal{I}$ such that

$\mathcal{I}(\texttt{Q}_1) := 4,\ \mathcal{I}(\texttt{Q}_2) := 8,\ \mathcal{I}(\texttt{Q}_3) := 1,\ \mathcal{I}(\texttt{Q}_4) := 2,\ \mathcal{I}(\texttt{Q}_5) := 6,\ \mathcal{I}(\texttt{Q}_6) := 2,\ \mathcal{I}(\texttt{Q}_7) := 7,\ \mathcal{I}(\texttt{Q}_8) := 5$,

then it is easy to see that this assignment is a solution of the eight queens problem. This solution is shown in Figure 7.13 on page 151.

Later, when we implement procedures to solve CSPs, we will represent variable assignments and partial variable assignments as dictionaries. For example, the variable assignment $\mathcal{I}$ defined above would then be represented as the dictionary

$\mathcal{I} = \bigl\{\texttt{Q}_1 : 4,\ \texttt{Q}_2 : 8,\ \texttt{Q}_3 : 1,\ \texttt{Q}_4 : 2,\ \texttt{Q}_5 : 6,\ \texttt{Q}_6 : 2,\ \texttt{Q}_7 : 7,\ \texttt{Q}_8 : 5)\bigr\}$.

If we define

$\mathcal{B} := \bigl\{\texttt{Q}_1 : 4,\ \texttt{Q}_2 : 8,\ \texttt{Q}_3 : 1)\bigr\}$,

then $\mathcal{B}$ is a partial assignment and $\texttt{dom}(\mathcal{B}) = \{\texttt{Q}_1, \texttt{Q}_2, \texttt{Q}_3\}$. This partial assignment is shown in Figure 7.14 on page 151.

Figure 7.15 on page 152 shows a *Python* program that can be used to create the eight queens puzzle as a CSP.

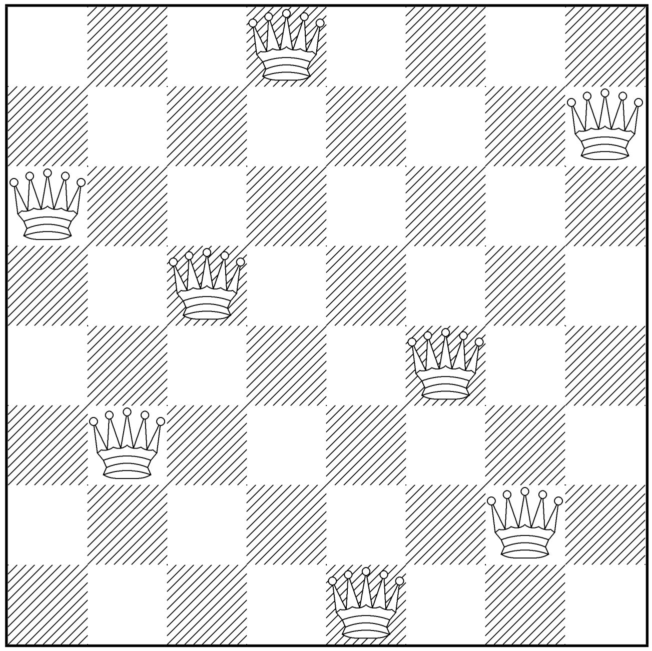

Figure 7.13: A solution of the eight queens problem.

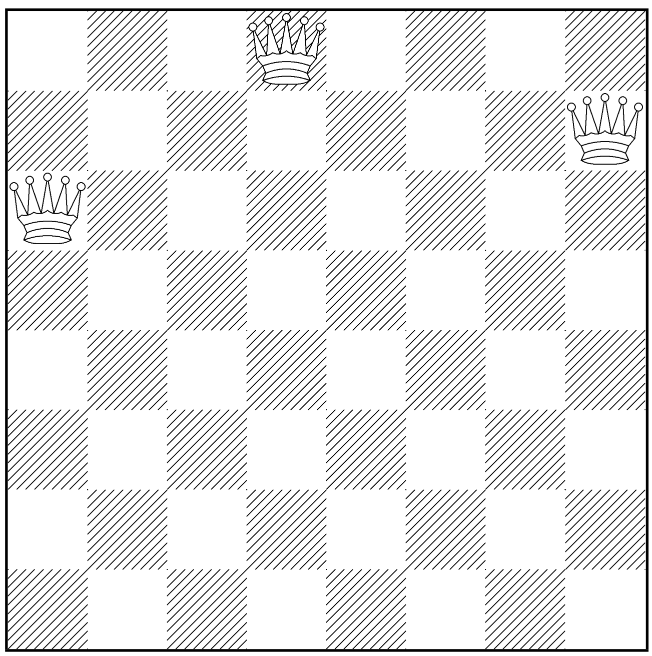

Figure 7.14: The partial assignment $\{\texttt{Q}_1, 4, \texttt{Q}_2, 8, \texttt{Q}_3, 1)\}$.

7.4.4 A Backtracking Constraint Solver

One approach to solve a CSP that is both conceptually simple and reasonable efficient is backtracking. The idea is to try to build variable assignments incrementally: We start with an empty dictionary and pick a variable x_1 that needs to have a value assigned. For this variable, we choose a value v_1 and assign it to this variable. This yields the partial assignment $\{x_1 : v_1\}$. Next, we evaluate all those

```
def queensCSP():
    "Returns a CSP coding the 8 queens problem."
    S            = range(1, 8+1)          # used as indices
    Variables    = [ f'Q{i}' for i in S ]
    Values       = { 1, 2, 3, 4, 5, 6, 7, 8 }
    SameRow      = { f'Q{i} != Q{j}' for i in S for j in S if i < j }
    SameDiagonal = { f'abs(Q{i}-Q{j}) != {j-i}' for i in S for j in S if i < j }
    return (Variables, Values, SameRow | SameDiagonal)
```

Figure 7.15: *Python* code to create the CSP representing the eight queens puzzle.

constraints that mention only the variable x_1 and check whether these constraints are satisfied. If any of these constraints is evaluated as `False`, we try to assign another value to x_1 until we find a value that satisfies all constraints that mention only x_1.

In general, if we have a partial variable assignment $\mathcal{B}$ of the form

$$\mathcal{B} = \{x_1 : v_1, \cdots, x_k : v_k\}$$

and we already know that all constraints that mention only the variables $x_1, \cdots, x_k$ are satisfied by $\mathcal{B}$, then in order to extend $\mathcal{B}$ we pick another variable x_{k+1} and choose a value v_{k+1} such that all those constraints that mention only the variables $x_1, \cdots, x_k, x_{k+1}$ are satisfied. If we discover that there is no such value v_{k+1}, then we have to undo the assignment $x_k : v_k$ and try to find a new value v_k such that, first, those constraints mentioning only the variables $x_1, \cdots, x_k$ are satisfied, and, second, it is possible to find a value v_{k+1} that can be assigned to x_{k+1}. This step of going back and trying to find a new value for the variable x_k is called backtracking. It might be necessary to backtrack more than one level and to also undo the assignment of v_{k-1} to x_{k-1} or, indeed, we might be forced to undo the assignments of all variables $x_i, \cdots, x_k$ for some $i \in \{1, \cdots, n\}$. The details of this search procedure are best explained by looking at its implementation. Figure 7.16 on page 152 shows a simple CSP solver that employs backtracking. We discuss this program next.

```
import extractVars as ev

def solve(CSP):
    "Compute a solution for the given constraint satisfaction problem."
    Variables, Values, Constraints = CSP
    CSP = (Variables,
           Values,
           [(f, ev.extractVars(f) & set(Variables)) for f in Constraints]
          )
    try:
        return backtrack_search({}, CSP)
    except Backtrack:
        return  # no solution found
```

Figure 7.16: A backtracking CSP solver

1. As we need to determine the variables occurring in a given constraint, we import the module `extractVar`. This module implements a function `extractVars`(f) that takes a formula f written as a *Python* expression and returns the set of all variables occurring in f.

2. The procedure `solve` takes a constraint satisfaction problem `CSP` as input and tries to find a solution.

   (a) First, in line 5 the `CSP` is split into its three components. However, the first component `Variables` does not have to be a set but rather can also be a list. If `Variables` is a list, then backtracking search will assign these variables in the same order as they appear in this list. This can improve the efficiency of backtracking significantly.

   (b) Next, for every constraint `f` of the given `CSP`, we compute the set of variables that are used in `f`. This is done using the procedure `extractVars`. Of these variables we keep only those variables that also occur in the set `Variables` because we assume that any other *Python* variable occurring in a constraint f has already a value assigned to it and can therefore be regarded as a constant.

   The variables occurring in a constraint `f` are then paired with the constraint `f` and the correspondingly modified data structure is stored in `CSP` and is called an augmented CSP.

   The reason to compute and store these variables is efficiency: When we later check whether a constraint `f` is satisfied for a partial variable assignment `Assignment` where `Assignment` is stored as a dictionary, we only need to check the constraint `f` iff all of the variables occurring in `f` are elements of the domain of `Assignment`. It would be wasteful to compute these variables every time.

   (c) Next, we call the function `backtrack_search` to compute a solution of `CSP`. This function call is enclosed in a `try-except`-block. The function `backtrack_search` either returns a solution or, if it is not able to find a solution, it throws an exception of class `Backtrack`. If this happens, the `except` block silently discards this exception and the procedure `solve` returns without a result.

Next, we discuss the implementation of the procedure `backtrack_search` that is shown in Figure 7.17 on page 154. This procedure receives a partial assignment `Assignment` as input together with an augmented `CSP`. This partial assignment is consistent with `CSP`: If `f` is a constraint of `CSP` such that all the variables occurring in `f` are assigned to in `Assignment`, then evaluating `f` using `Assignment` yields `True`. Initially, this partial assignment is empty and hence trivially consistent. The idea is to extend this partial assignment until it is a complete assignment that satisfies all constraints of the given `CSP`.

1. First, the augmented `CSP` is split into its components.

2. Next, if `Assignment` is already a complete variable assignment, i.e. if the dictionary `Assignment` has as many elements as there are variables, then it must be a solution of the `CSP` and, therefore, it is returned.

3. Otherwise, we have to extend the partial `Assignment`. In order to do so, we first have to select a variable `var` that has not yet been assigned a value in `Assignment` so far. We pick the first variable in the list `Variables` that is yet unassigned. This variable is called `var`.

4. Next, it is tried to assign a `value` to the selected variable `var`. After assigning a `value` to `var`, we immediately check whether this assignment would be consistent with the constraints using the procedure `is_consistent`. If the partial `Assignment` turns out to be consistent, the partial

```
def backtrack_search(Assignment, CSP):
    """
    Given a partial variable assignment, this function tries to complete this assignment
    towards a solution of the CSP.
    """
    (Variables, Values, Constraints) = CSP
    if len(Assignment) == len(Variables):
        return Assignment
    var = [x for x in Variables if x not in Assignment][0]
    for value in Values:
        try:
            if isConsistent(var, value, Assignment, Constraints):
                NewAssign      = Assignment.copy()
                NewAssign[var] = value
                return backtrack_search(NewAssign, CSP)
        except Backtrack:
            continue
    # all values have been tried without success, no solution has been found
    raise Backtrack()

class Backtrack(Exception):
    pass
```

Figure 7.17: The function `backtrack_search`

`Assignment` is extended to the new partial assignment `NewAssign` that satisfies

```
NewAssign[var] = value
```

and that coincides with `Assignment` for all variables different from `var`. Then, the procedure `backtrack_search` is called recursively to complete this new partial assignment. If this is successful, the resulting assignment is a solution of the CSP and is returned. Otherwise, the recursive call of `backtrack_search` will instead raise an exception. This exception is muted by the `try`-`except`-block that surrounds the recursive call to `backtrack_search`. In that case, the `for`-loop generates a new possible `value` that can be assigned to the variable `var`. If all possible values have been tried and none was successful, the `for`-loop ends and the statement

```
raise Backtrack()
```

is executed. This raises an exception that signals that the current partial `Assignment` can not be completed into a solution of the `CSP`. This exception is caught by one of the `try`-`except`-blocks that have been encountered previously.

We still need to discuss the implementation of the auxiliary procedure `isConsistent` shown in Figure 7.18 on page 155. This procedure takes a variable `var`, a `value`, a partial `Assignment` and a set of `Constraints`. It is assumed that `Assignment` is partially consistent with respect to the set `Constraints`, i.e. for every formula `f` occurring in `Constraints` such that

$$\texttt{vars}(\texttt{f}) \subseteq \texttt{dom}(\texttt{Assignment})$$

```
def isConsistent(var, value, Assignment, Constraints):
    NewAssign      = Assignment.copy()
    NewAssign[var] = value
    return all(eval(f, NewAssign) for (f, Vs) in Constraints
                                  if var in Vs and Vs <= NewAssign.keys()
              )
```

Figure 7.18: The procedure `isConsistent`

holds, the formula `f` evaluates to `True` given the `Assignment`. The purpose of `isConsistent` is to check, whether the extended assignment

$$\texttt{NA} := \texttt{Assignment} \cup \{\langle \texttt{var}, \texttt{value} \rangle\}$$

that assigns `value` to the variable `var` is still partially consistent with `Constraints`. To this end, the `for`-loop iterates over all `Formula` in `Constraints`. However, we only have to check those `Formula` that contain the variable `var` and, furthermore, have the property that

$$\texttt{Vars}(\texttt{Formula}) \subseteq \texttt{dom}(\texttt{NA}),$$

i.e. all variables occurring in `Formula` need to have a value assigned in `NA`. The reasoning is as follows:

1. If `var` does not occur in `Formula`, then adding `var` to `Assignment` cannot change the result of evaluating `Formula` and as `Assignment` is assumed to be partially consistent with respect to `Formula`, `NA` is also partially consistent with respect to `Formula`.
2. If $\texttt{dom}(\texttt{NA}) \not\subseteq \texttt{Vars}(\texttt{Formula})$, then `Formula` can not be evaluated anyway.

If we use backtracking, we can solve the 8 queens problem in less than a second. For the eight queens puzzle the order in which variables are tried is not particularly important. The reason is that all variables are connected to all other variables. For other problems the ordering of the variables can be very important. The general strategy is that variables that are strongly related to each other should be grouped together in the list `Variables`.

7.5 Normalformen für prädikatenlogische Formeln

Im nächsten Abschnitt gehen wir daran, einen Kalkül $\vdash$ für die Prädikaten-Logik zu definieren. Genau wie im Falle der Aussagen-Logik wird dies wesentlich einfacher, wenn wir uns auf Formeln, die in einer *Normalform* vorliegen, beschränken. Bei dieser Normalform handelt es sich nun um sogenannte prädikatenlogische Klauseln. Wir zeigen in diesem Abschnitt, dass jede Formel-Menge M so in eine Menge von Klauseln K transformiert werden kann, dass M genau dann erfüllbar ist, wenn K erfüllbar ist. Daher ist die Beschränkung auf Klauseln keine echte Einschränkung. Zunächst geben wir einige Äquivalenzen an, mit deren Hilfe Quantoren manipuliert werden können.

Satz 40 *Es gelten die folgenden Äquivalenzen:*

1. $\models \neg(\forall x\colon f) \leftrightarrow (\exists x\colon \neg f)$
2. $\models \neg(\exists x\colon f) \leftrightarrow (\forall x\colon \neg f)$

3. $\models \forall x\colon f \wedge \forall x\colon g \leftrightarrow \forall x\colon (f \wedge g)$

4. $\models \exists x\colon f \vee \exists x\colon g \leftrightarrow \exists x\colon (f \vee g)$

5. $\models \forall x\colon \forall y\colon f \leftrightarrow \forall y\colon \forall x\colon f$

6. $\models \exists x\colon \exists y\colon f \leftrightarrow \exists y\colon \exists x\colon f$

7. *Falls x eine Variable ist, für die $x \notin FV(f)$ ist, so haben wir*
$$\models \forall x\colon f \leftrightarrow f \quad \textit{und} \quad \models \exists x\colon f \leftrightarrow f.$$

8. *Falls x eine Variable ist, für die $x \notin FV(g)$ gilt, so haben wir die folgenden Äquivalenzen:*

   (a) $\models (\forall x\colon f) \vee g \leftrightarrow \forall x\colon (f \vee g) \quad \textit{und} \quad \models g \vee (\forall x\colon f) \leftrightarrow \forall x\colon (g \vee f),$
   (b) $\models (\exists x\colon f) \wedge g \leftrightarrow \exists x\colon (f \wedge g) \quad \textit{und} \quad \models g \wedge (\exists x\colon f) \leftrightarrow \exists x\colon (g \wedge f).$

Um die Äquivalenzen der letzten Gruppe anwenden zu können, kann es notwendig sein, gebundene Variablen umzubenennen. Ist f eine prädikatenlogische Formel und sind x und y zwei Variablen, so bezeichnet $f[x/y]$ die Formel, die aus f dadurch entsteht, dass jedes Auftreten der Variablen x in f durch y ersetzt wird. Beispielsweise gilt
$$\big(\forall u : \exists v : p(u,v)\big)[u/z] = \forall z : \exists v : p(z,v)$$
Damit können wir eine letzte Äquivalenz angeben: Ist f eine prädikatenlogische Formel, ist $x \in BV(f)$ und ist y eine Variable, die in f nicht auftritt, so gilt
$$\models f \leftrightarrow f[x/y].$$

Mit Hilfe der oben stehenden Äquivalenzen und der aussagenlogischen Äquivalenzen, die wir schon kennen, können wir eine Formel so umformen, dass die Quantoren nur noch außen stehen. Eine solche Formel ist dann in pränexer Normalform. Wir führen das Verfahren an einem Beispiel vor: Wir zeigen, dass die Formel
$$\forall x\colon p(x) \rightarrow \exists x\colon p(x)$$
allgemeingültig ist:
$$\begin{array}{ll} & \forall x\colon p(x) \rightarrow \exists x\colon p(x) \\ \Leftrightarrow & \neg\forall x\colon p(x) \vee \exists x\colon p(x) \\ \Leftrightarrow & \exists x\colon \neg p(x) \vee \exists x\colon p(x) \\ \Leftrightarrow & \exists x\colon \big(\neg p(x) \vee p(x)\big) \\ \Leftrightarrow & \exists x\colon \top \\ \Leftrightarrow & \top \end{array}$$

In diesem Fall haben wir Glück gehabt, dass es uns gelungen ist, die Formel als Tautologie zu erkennen. Im Allgemeinen reichen die obigen Umformungen aber nicht aus, um prädikatenlogische Tautologien erkennen zu können. Um Formeln noch stärker vereinfachen zu können, führen wir einen weiteren Äquivalenz-Begriff ein. Diesen Begriff wollen wir vorher durch ein Beispiel motivieren. Wir betrachten die beiden Formeln
$$f_1 = \forall x\colon \exists y\colon p(x,y) \quad \text{und} \quad f_2 = \forall x\colon p\big(x, s(x)\big).$$
Die beiden Formeln f_1 und f_2 sind nicht äquivalent, denn sie entstammen noch nicht einmal der gleichen Signatur: In der Formel f_2 wird das Funktions-Zeichen s verwendet, das in der Formel f_1

überhaupt nicht auftritt. Auch wenn die beiden Formeln f_1 und f_2 nicht äquivalent sind, so besteht zwischen ihnen doch die folgende Beziehung: Ist S_1 eine prädikatenlogische Struktur, in der die Formel f_1 gilt:

$$\mathcal{S}_1 \models f_1,$$

dann können wir diese Struktur zu einer Struktur S_2 erweitern, in der die Formel f_2 gilt:

$$\mathcal{S}_2 \models f_2.$$

Dazu muss lediglich die Interpretation des Funktions-Zeichens s so gewählt werden, dass für jedes x tatsächlich $p\big(x, s(x)\big)$ gilt. Dies ist möglich, denn die Formel f_1 sagt ja aus, dass wir zu jedem x einen Wert y finden, für den $p(x, y)$ gilt. Die Funktion s muss also lediglich zu jedem x dieses y zurück geben.

Definition 41 (Skolemisierung) *Es sei $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \textit{arity} \rangle$ eine Signatur. Ferner sei f eine geschlossene Σ-Formel der Form*

$$f = \forall x_1, \cdots, x_n \colon \exists y \colon g.$$

Dann wählen wir ein neues n-stelliges Funktions-Zeichen s, d.h. wir nehmen ein Zeichen s, dass in der Signatur Σ nicht auftritt und erweitern die Signatur Σ zu der Signatur

$$\Sigma' := \Big\langle \mathcal{V}, \mathcal{F} \cup \{s\}, \mathcal{P}, \textit{arity} \cup \big\{\langle s, n \rangle\big\} \Big\rangle,$$

in der wir s als neues n-stelliges Funktions-Zeichen deklarieren. Anschließend definieren wir die Σ'-Formel f' wie folgt:

$$f' := \texttt{Skolem}(f) := \forall x_1 \colon \cdots \forall x_n \colon g\big[y \mapsto s(x_1, \cdots, x_n)\big]$$

Hierbei bezeichnet der Ausdruck $g\big[y \mapsto s(x_1, \cdots, x_n)\big]$ die Formel, die wir aus g dadurch erhalten, dass wir jedes Auftreten der Variablen y in der Formel g durch den Term $s(x_1, \cdots, x_n)$ ersetzen. Wir sagen, dass die Formel f' aus der Formel f durch einen Skolemisierungs-Schritt hervorgegangen ist. ◇

Beispiel: Es f die folgende Formel aus der Gruppen-Theorie:

$$f := \forall x : \exists y : y * x = 1.$$

Dann gilt

$$\texttt{Skolem}(f) = \forall x : s(x) * x = 1.$$ ◇

In welchem Sinne sind eine Formel f und eine Formel f', die aus f durch einen Skolemisierungs-Schritt hervorgegangen sind, äquivalent? Zur Beantwortung dieser Frage dient die folgende Definition.

Definition 42 (Erfüllbarkeits-Äquivalenz)
Zwei geschlossene Formeln f und g heißen erfüllbarkeits-äquivalent falls f und g entweder beide erfüllbar oder beide unerfüllbar sind. Wenn f und g erfüllbarkeits-äquivalent sind, so schreiben wir

$$f \approx_e g.$$ ◇

Beobachtung: Falls die Formel f' aus der Formel f durch einen Skolemisierungs-Schritt hervorgegangen ist, so sind f und f' erfüllbarkeits-äquivalent, denn wir können jede Struktur $\mathcal{S}$, in der die Formel f gilt, zu einer Struktur $\mathcal{S}'$ erweitern, in der auch f' gilt. ◇

Wir können nun ein einfaches Verfahren angeben, um Existenz-Quantoren aus einer Formel zu eliminieren. Dieses Verfahren besteht aus zwei Schritten: Zunächst bringen wir die Formel in pränexe Normalform. Anschließend können wir die Existenz-Quantoren der Reihe nach durch Skolemisierungs-Schritte eliminieren. Nach dem eben gezeigten Satz ist die resultierende Formel zu der ursprünglichen Formel erfüllbarkeits-äquivalent. Dieses Verfahren der Eliminierung von Existenz-Quantoren durch die Einführung neuer Funktions-Zeichen wird als Skolemisierung bezeichnet. Haben wir eine Formel F in pränexe Normalform gebracht und anschließend skolemisiert, so hat das Ergebnis die Gestalt

$$\forall x_1, \cdots, x_n : g$$

und in der Formel g treten keine Quantoren mehr auf. Die Formel g wird auch als die Matrix der obigen Formel bezeichnet. Wir können nun g mit Hilfe der uns aus dem letzten Kapitel bekannten aussagenlogischen Äquivalenzen in konjunktive Normalform bringen. Wir haben dann eine Formel der Gestalt

$$\forall x_1, \cdots, x_n : (k_1 \wedge \cdots \wedge k_m).$$

Dabei sind die k_i Disjunktionen von Literalen. In der Prädikatenlogik ist ein Literal entweder eine atomare Formel oder die Negation einer atomaren Formel. Wenden wir hier die Äquivalenz

$$\forall x\colon (f_1 \wedge f_2) \leftrightarrow (\forall x\colon f_1) \wedge (\forall x\colon f_2)$$

an, so können wir die All-Quantoren auf die einzelnen k_i verteilen und die resultierende Formel hat die Gestalt

$$\big(\forall x_1, \cdots, x_n : k_1\big) \wedge \cdots \wedge \big(\forall x_1, \cdots, x_n : k_m\big).$$

Ist eine Formel F in der obigen Gestalt, so sagen wir, dass F in prädikatenlogischer Klausel-Normalform ist und eine Formel der Gestalt

$$\forall x_1, \cdots, x_n : k,$$

bei der k eine Disjunktion prädikatenlogischer Literale ist, bezeichnen wir als prädikatenlogische Klausel. Ist M eine Menge von Formeln deren Erfüllbarkeit wir untersuchen wollen, so können wir nach dem bisher gezeigten M immer in eine Menge prädikatenlogischer Klauseln umformen. Da dann nur noch All-Quantoren vorkommen, können wir hier die Notation noch vereinfachen, indem wir vereinbaren, dass alle Formeln implizit allquantifiziert sind, wir lassen also die All-Quantoren weg.

Wozu sind nun die Umformungen in Skolem-Normalform gut? Es geht darum, dass wir ein Verfahren entwickeln wollen, mit dem es möglich ist für eine prädikatenlogische Formel f zu zeigen, dass f allgemeingültig ist, dass also

$$\models f$$

gilt. Wir wissen, dass

$$\models f \quad \text{g.d.w.} \quad \{\neg f\} \models \bot$$

gilt, denn die Formel f ist genau dann allgemeingültig, wenn es keine Struktur gibt, in der die Formel $\neg f$ erfüllbar ist. Wir bilden daher zunächst $\neg f$ und formen $\neg f$ in prädikatenlogische Klausel-Normalform um. Wir erhalten Klauseln $k_1, \cdots, k_n$, so dass

$$\neg f \approx_e k_1 \wedge \cdots \wedge k_n$$

gilt. Anschließend versuchen wir, aus den Klauseln $k_1, \cdots, k_n$ einen Widerspruch herzuleiten:

$$\{k_1, \cdots, k_n\} \vdash \bot$$

Wenn dies gelingt, dann wissen wir, dass die Menge $\{k_1, \cdots, k_n\}$ unerfüllbar ist. Damit ist auch $\neg f$ unerfüllbar und also ist f allgemeingültig. Damit wir aus den Klauseln $k_1, \cdots, k_n$ einen Widerspruch herleiten können, brauchen wir natürlich noch einen Kalkül, der mit prädikatenlogischen Klauseln arbeitet. Einen solchen Kalkül werden wir im übernächsten Abschnitt vorstellen.

Um das Verfahren näher zu erläutern demonstrieren wir es an einem Beispiel. Wir wollen untersuchen, ob

$$\models (\exists x\colon \forall y\colon p(x,y)) \rightarrow (\forall y\colon \exists x\colon p(x,y))$$

gilt. Wir wissen, dass dies äquivalent dazu ist, dass

$$\Big\{\neg\Big((\exists x\colon \forall y\colon p(x,y)) \rightarrow (\forall y\colon \exists x\colon p(x,y))\Big)\Big\} \models \bot$$

gilt. Wir bringen zunächst die negierte Formel in pränexe Normalform.

$$\begin{array}{ll}
 & \neg\Big((\exists x\colon \forall y\colon p(x,y)) \rightarrow (\forall y\colon \exists x\colon p(x,y))\Big) \\
\leftrightarrow & \neg\Big(\neg(\exists x\colon \forall y\colon p(x,y)) \vee (\forall y\colon \exists x\colon p(x,y))\Big) \\
\leftrightarrow & (\exists x\colon \forall y\colon p(x,y)) \wedge \neg(\forall y\colon \exists x\colon p(x,y)) \\
\leftrightarrow & (\exists x\colon \forall y\colon p(x,y)) \wedge (\exists y\colon \neg\exists x\colon p(x,y)) \\
\leftrightarrow & (\exists x\colon \forall y\colon p(x,y)) \wedge (\exists y\colon \forall x\colon \neg p(x,y))
\end{array}$$

Um an dieser Stelle weitermachen zu können, ist es nötig, die Variablen in dem zweiten Glied der Konjunktion umzubenennen. Wir ersetzen x durch u und y durch v und erhalten

$$\begin{array}{ll}
 & (\exists x\colon \forall y\colon p(x,y)) \wedge (\exists y\colon \forall x\colon \neg p(x,y)) \\
\leftrightarrow & (\exists x\colon \forall y\colon p(x,y)) \wedge (\exists v\colon \forall u\colon \neg p(u,v)) \\
\leftrightarrow & \exists v\colon \Big((\exists x\colon \forall y\colon p(x,y)) \wedge (\forall u\colon \neg p(u,v))\Big) \\
\leftrightarrow & \exists v\colon \exists x\colon \Big((\forall y\colon p(x,y)) \wedge (\forall u\colon \neg p(u,v))\Big) \\
\leftrightarrow & \exists v\colon \exists x\colon \forall y\colon \Big(p(x,y) \wedge (\forall u\colon \neg p(u,v))\Big) \\
\leftrightarrow & \exists v\colon \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,v)\Big)
\end{array}$$

An dieser Stelle müssen wir skolemisieren um die Existenz-Quantoren los zu werden. Wir führen dazu zwei neue Funktions-Zeichen s_1 und s_2 ein. Dabei gilt $\texttt{arity}(s_1) = 0$ und $\texttt{arity}(s_2) = 0$, denn vor den Existenz-Quantoren stehen keine All-Quantoren.

$$\begin{array}{ll}
 & \exists v\colon \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,v)\Big) \\
\approx_e & \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,s_1)\Big) \\
\approx_e & \forall y\colon \forall u\colon \Big(p(s_2,y) \wedge \neg p(u,s_1)\Big)
\end{array}$$

Da jetzt nur noch All-Quantoren auftreten, können wir diese auch noch weglassen, da wir ja vereinbart haben, dass alle freien Variablen implizit allquantifiziert sind. Damit können wir nun die prädikatenlogische Klausel-Normalform in Mengen-Schreibweise angeben, diese ist

$$M := \Big\{\{p(s_2,y)\}, \{\neg p(u,s_1)\}\Big\}.$$

Wir zeigen, dass die Menge M widersprüchlich ist. Dazu betrachten wir zunächst die Klausel

$\{p(s_2,y)\}$ und setzen in dieser Klausel für y die Konstante s_1 ein. Damit erhalten wir die Klausel

$$\{p(s_2,s_1)\}. \tag{1}$$

Das Ersetzung von y durch s_1 begründen wir damit, dass die obige Klausel ja implizit allquantifiziert ist und wenn etwas für alle y gilt, dann sicher auch für $y = s_1$.

Als nächstes betrachten wir die Klausel $\{\neg p(u,s_1)\}$. Hier setzen wir für die Variablen u die Konstante s_2 ein und erhalten dann die Klausel

$$\{\neg p(s_2,s_1)\} \tag{2}$$

Nun wenden wir auf die Klauseln (1) und (2) die Schnitt-Regel an und finden

$$\{p(s_2,s_1)\}, \quad \{\neg p(s_2,s_1)\} \quad \vdash \quad \{\}.$$

Damit haben wir einen Widerspruch hergeleitet und gezeigt, dass die Menge M unerfüllbar ist. Damit ist dann auch

$$\Big\{\neg\Big((\exists x\colon \forall y\colon p(x,y)) \rightarrow (\forall y\colon \exists x\colon p(x,y))\Big)\Big\}$$

unerfüllbar und folglich gilt

$$\models (\exists x\colon \forall y\colon p(x,y)) \rightarrow (\forall y\colon \exists x\colon p(x,y)).$$

7.6 Unifikation

In dem Beispiel im letzten Abschnitt haben wir die Terme s_1 und s_2 geraten, die wir für die Variablen y und u in den Klauseln $\{p(s_2,y)\}$ und $\{\neg p(u,s_1)\}$ eingesetzt haben. Wir haben diese Terme mit dem Ziel gewählt, später die Schnitt-Regel anwenden zu können. In diesem Abschnitt zeigen wir nun ein Verfahren, mit dessen Hilfe wir die benötigten Terme ausrechnen können. Dazu benötigen wir zunächst den Begriff einer Substitution.

Definition 43 (Substitution) *Es sei eine Signatur*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$$

gegeben. Eine Σ-Substitution ist eine endliche Menge von Paaren der Form

$$\sigma = \{\langle x_1, t_1\rangle, \cdots, \langle x_n, t_n\rangle\}.$$

Dabei gilt:

1. *$x_i \in \mathcal{V}$, die x_i sind also Variablen.*
2. *$t_i \in \mathcal{T}_\Sigma$, die t_i sind also Terme.*
3. *Für $i \neq j$ ist $x_i \neq x_j$, die Variablen sind also paarweise verschieden.*

Ist $\sigma = \{\langle x_1, t_1\rangle, \cdots, \langle x_n, t_n\rangle\}$ eine Σ-Substitution, so schreiben wir

$$\sigma = [x_1 \mapsto t_1, \cdots, x_n \mapsto t_n].$$

Außerdem definieren wir den Domain einer Substitution als

$$dom(\sigma) := \{x_1, \cdots, x_n\}.$$

Die Menge aller Substitutionen bezeichnen wir mit Subst. ◇

Substitutionen werden für uns dadurch interessant, dass wir sie auf Terme anwenden können. Ist t ein Term und σ eine Substitution, so ist $t\sigma$ der Term, der aus t dadurch entsteht, dass jedes Vorkommen einer Variablen x_i durch den zugehörigen Term t_i ersetzt wird. Die formale Definition folgt.

Definition 44 (Anwendung einer Substitution)
Es sei t ein Term und es sei $\sigma = [x_1 \mapsto t_1, \cdots, x_n \mapsto t_n]$ eine Substitution. Wir definieren die Anwendung von σ auf t (Schreibweise $t\sigma$) durch Induktion über den Aufbau von t:

1. *Falls t eine Variable ist, gibt es zwei Fälle:*
   (a) *$t = x_i$ für ein $i \in \{1, \cdots, n\}$. Dann definieren wir* $x_i\sigma := t_i$.
   (b) *$t = y$ mit $y \in \mathcal{V}$, aber $y \notin \{x_1, \cdots, x_n\}$. Dann definieren wir* $y\sigma := y$.
2. *Andernfalls muss t die Form $t = f(s_1, \cdots, s_m)$ haben. Dann können wir $t\sigma$ durch*
   $$f(s_1, \cdots, s_m)\sigma := f(s_1\sigma, \cdots, s_m\sigma).$$
   definieren, denn nach Induktions-Voraussetzung sind die Ausdrücke $s_i\sigma$ bereits definiert. $\diamond$

Genau wie wir Substitutionen auf Terme anwenden können, können wir eine Substitution auch auf prädikatenlogische Klauseln anwenden. Dabei werden Prädikats-Zeichen und Junktoren wie Funktions-Zeichen behandelt. Wir ersparen uns eine formale Definition und geben stattdessen zunächst einige Beispiele. Wir definieren eine Substitution σ durch

$$\sigma := [x_1 \mapsto c,\ x_2 \mapsto f(d)].$$

In den folgenden drei Beispielen demonstrieren wir zunächst, wie eine Substitution auf einen Term angewendet werden kann. Im vierten Beispiel wenden wir die Substitution dann auf eine Klausel an:

1. $x_3\sigma = x_3$,
2. $f(x_2)\sigma = f\big(f(d)\big)$,
3. $h(x_1, g(x_2))\sigma = h\big(c, g(f(d))\big)$.
4. $\big\{p(x_2), q(d, h(x_3, x_1))\big\}\sigma = \big\{p(f(d)),\ q(d, h(x_3, c))\big\}$.

Als nächstes zeigen wir, wie Substitutionen miteinander verknüpft werden können.

Definition 45 (Komposition von Substitutionen) *Es seien*

$$\sigma = [x_1 \mapsto s_1, \cdots, x_m \mapsto s_m] \quad \text{und} \quad \tau = [y_1 \mapsto t_1, \cdots, y_n \mapsto t_n]$$

zwei Substitutionen mit $dom(\sigma) \cap dom(\tau) = \{\}$. Dann definieren wir die Komposition $\sigma\tau$ von σ und τ als

$$\sigma\tau := [x_1 \mapsto s_1\tau, \cdots, x_m \mapsto s_m\tau,\ y_1 \mapsto t_1, \cdots, y_n \mapsto t_n]$$ $\diamond$

Beispiel: Wir führen das obige Beispiel fort und setzen

$$\sigma := [x_1 \mapsto c,\ x_2 \mapsto f(x_3)] \quad \text{und} \quad \tau := [x_3 \mapsto h(c,c),\ x_4 \mapsto d].$$

Dann gilt:

$$\sigma\tau = \big[x_1 \mapsto c,\ x_2 \mapsto f(h(c,c)),\ x_3 \mapsto h(c,c),\ x_4 \mapsto d\big]. \qquad \square$$

Die Definition der Komposition von Substitutionen ist mit dem Ziel gewählt worden, dass der folgende Satz gilt.

Satz 46 *Ist t ein Term und sind σ und τ Substitutionen mit $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\tau) = \{\}$, so gilt*

$$(t\sigma)\tau = t(\sigma\tau). \qquad \square$$

Der Satz kann durch Induktion über den Aufbau des Termes t bewiesen werden.

Definition 47 (Syntaktische Gleichung) *Unter einer syntaktischen Gleichung verstehen wir in diesem Abschnitt ein Konstrukt der Form $s \doteq t$, wobei einer der beiden folgenden Fälle vorliegen muss:*

1. *s und t sind Terme oder*
2. *s und t sind atomare Formeln.*

Weiter definieren wir ein syntaktisches Gleichungs-System als eine Menge von syntaktischen Gleichungen. ◇

Was syntaktische Gleichungen angeht, so machen wir keinen Unterschied zwischen Funktions-Zeichen und Prädikats-Zeichen. Dieser Ansatz ist deswegen berechtigt, weil wir Prädikate ja auch als spezielle Funktionen auffassen können, nämlich als solche Funktionen, die als Ergebnis einen Wahrheitswert aus der Menge $\mathbb{B}$ zurück geben.

Definition 48 (Unifikator) *Eine Substitution σ löst eine syntaktische Gleichung $s \doteq t$ genau dann, wenn $s\sigma = t\sigma$ ist, wenn also durch die Anwendung von σ auf s und t tatsächlich identische Objekte entstehen. Ist E ein syntaktisches Gleichungs-System, so sagen wir, dass σ ein Unifikator von E ist wenn σ jede syntaktische Gleichung in E löst.* ◇

Ist $E = \{s_1 \doteq t_1, \cdots, s_n \doteq t_n\}$ eine syntaktisches Gleichungs-System und ist σ eine Substitution, so definieren wir

$$E\sigma := \{s_1\sigma \doteq t_1\sigma, \cdots, s_n\sigma \doteq t_n\sigma\}.$$

Beispiel: Wir verdeutlichen die bisher eingeführten Begriffe anhand eines Beispiels. Wir betrachten die Gleichung

$$p(x_1, f(x_4)) \doteq p(x_2, x_3)$$

und definieren die Substitution

$$\sigma := \big[x_1 \mapsto x_2,\ x_3 \mapsto f(x_4)\big].$$

Die Substitution σ löst die obige syntaktische Gleichung, denn es gilt

$$\begin{aligned} p(x_1, f(x_4))\sigma &= p(x_2, f(x_4)) \quad \text{und} \\ p(x_2, x_3)\sigma &= p(x_2, f(x_4)). \end{aligned} \qquad \diamond$$

Als nächstes entwickeln wir ein Verfahren, mit dessen Hilfe wir von einer vorgegebenen Menge E von syntaktischen Gleichungen entscheiden können, ob es einen Unifikator σ für E gibt. Das Verfahren, das wir entwickeln werden, wurde von Martelli und Montanari veröffentlicht [MM82]. Wir überlegen uns zunächst, in welchen Fällen wir eine syntaktischen Gleichung $s \doteq t$ garantiert nicht lösen können. Da gibt es zwei Möglichkeiten: Eine syntaktische Gleichung

$$f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)$$

ist sicher dann nicht durch eine Substitution lösbar, wenn f und g verschiedene Funktions-Zeichen sind, denn für jede Substitution σ gilt ja

$$f(s_1, \cdots, s_m)\sigma = f(s_1\sigma, \cdots, s_m\sigma) \quad \text{und} \quad g(t_1, \cdots, t_n)\sigma = g(t_1\sigma, \cdots, t_n\sigma).$$

Falls $f \neq g$ ist, haben die Terme $f(s_1, \cdots, s_m)\sigma$ und $g(t_1, \cdots, t_n)\sigma$ verschieden Funktions-Zeichen und können daher syntaktisch nicht identisch werden.

Die andere Form einer syntaktischen Gleichung, die garantiert unlösbar ist, ist

$$x \doteq f(t_1, \cdots, t_n) \quad \text{falls } x \in \mathit{Var}\big(f(t_1, \cdots, t_n)\big).$$

Das diese syntaktische Gleichung unlösbar ist liegt daran, dass die rechte Seite immer mindestens ein Funktions-Zeichen mehr enthält als die linke.

Mit diesen Vorbemerkungen können wir nun ein Verfahren angeben, mit dessen Hilfe es möglich ist, Mengen von syntaktischen Gleichungen zu lösen, oder festzustellen, dass es keine Lösung gibt. Das Verfahren operiert auf Paaren der Form $\langle F, \tau \rangle$. Dabei ist F ein syntaktisches Gleichungs-System und τ ist eine Substitution. Wir starten das Verfahren mit dem Paar $\langle E, [] \rangle$. Hierbei ist E das zu lösende Gleichungs-System und $[]$ ist die leere Substitution. Das Verfahren arbeitet, indem die im Folgenden dargestellten Reduktions-Regeln solange angewendet werden, bis entweder feststeht, dass die Menge der Gleichungen keine Lösung hat, oder aber ein Paar der Form $\langle \{\}, \sigma \rangle$ erreicht wird. In diesem Fall ist σ ein Unifikator der Menge E, mit der wir gestartet sind. Es folgen die Reduktions-Regeln:

1. Falls $y \in \mathcal{V}$ eine Variable ist, die nicht in dem Term t auftritt, so können wir die folgende Reduktion durchführen:

   $$\Big\langle E \cup \big\{y \doteq t\big\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E[y \mapsto t], \sigma\big[y \mapsto t\big] \Big\rangle$$

   Diese Reduktions-Regel ist folgendermaßen zu lesen: Enthält die zu untersuchende Menge von syntaktischen Gleichungen eine Gleichung der Form $y \doteq t$, wobei die Variable y nicht in t auftritt, dann können wir diese Gleichung aus der gegebenen Menge von Gleichungen entfernen. Gleichzeitig wird die Substitution σ in die Substitution $\sigma\big[y \mapsto t\big]$ transformiert und auf die restlichen syntaktischen Gleichungen wird die Substitution $[y \mapsto t]$ angewendet.

2. Wenn die Variable y in dem Term t auftritt, falls also $y \in \mathit{Var}(t)$ ist und wenn außerdem $t \neq y$ ist, dann hat das Gleichungs-System $E \cup \big\{y \doteq t\big\}$ keine Lösung, wir schreiben

   $$\Big\langle E \cup \big\{y \doteq t\big\}, \sigma \Big\rangle \rightsquigarrow \Omega \quad \text{falls } x \in \mathit{Var}(t) \text{ und } y \neq t.$$

3. Falls $y \in \mathcal{V}$ eine Variable ist und t keine Variable ist, so haben wir folgende Reduktions-Regel:

   $$\Big\langle E \cup \big\{t \doteq y\big\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E \cup \big\{y \doteq t\big\}, \sigma \Big\rangle.$$

   Diese Regel wird benötigt, um anschließend eine der ersten beiden Regeln anwenden zu können.

4. Triviale syntaktische Gleichungen von Variablen können wir einfach weglassen:

$$\Big\langle E \cup \{x \doteq x\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E, \sigma \Big\rangle.$$

5. Ist f ein n-stelliges Funktions-Zeichen, so gilt

$$\Big\langle E \cup \{f(s_1, \cdots, s_n) \doteq f(t_1, \cdots, t_n)\}, \sigma \Big\rangle \rightsquigarrow \Big\langle E \cup \{s_1 \doteq t_1, \cdots, s_n \doteq t_n\}, \sigma \Big\rangle.$$

Eine syntaktische Gleichung der Form $f(s_1, \cdots, s_n) \doteq f(t_1, \cdots, t_n)$ wird also ersetzt durch die n syntaktische Gleichungen $s_1 \doteq t_1, \cdots, s_n \doteq t_n$.

Diese Regel ist im übrigen der Grund dafür, dass wir mit Mengen von syntaktischen Gleichungen arbeiten müssen, denn auch wenn wir mit nur einer syntaktischen Gleichung starten, kann durch die Anwendung dieser Regel die Zahl der syntaktischen Gleichungen erhöht werden.

Ein Spezialfall dieser Regel ist

$$\Big\langle E \cup \{c \doteq c\}, \sigma \Big\rangle \rightsquigarrow \Big\langle E, \sigma \Big\rangle.$$

Hier steht c für eine Konstante, also ein 0-stelliges Funktions-Zeichen. Triviale Gleichungen über Konstanten können also einfach weggelassen werden.

6. Das Gleichungs-System $E \cup \{f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)\}$ hat keine Lösung, falls die Funktions-Zeichen f und g verschieden sind, wir schreiben

$$\Big\langle E \cup \{f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)\}, \sigma \Big\rangle \rightsquigarrow \Omega \qquad \text{falls } f \neq g.$$

Haben wir ein nicht-leeres Gleichungs-System E gegeben und starten mit dem Paar $\langle E, [] \rangle$, so lässt sich immer eine der obigen Regeln anwenden. Diese geht solange bis einer der folgenden Fälle eintritt:

1. Die 2. oder die 6. Regel ist anwendbar. Dann hat das Gleichungs-System E keine Lösung und als Ergebnis der Unifikation wird Ω zurück gegeben.
2. Das Paar $\langle E, [] \rangle$ wird reduziert zu einem Paar $\langle \{\}, \sigma \rangle$. Dann ist σ ein Unifikator von E. In diesem Fall schreiben wir $\sigma = \mathtt{mgu}(E)$. Falls $E = \{s \doteq t\}$ ist, schreiben wir auch $\sigma = \mathtt{mgu}(s, t)$. Die Abkürzung `mgu` steht hier für "most general unifier".

Beispiel: Wir wenden das oben dargestellte Verfahren an, um die syntaktische Gleichung

$$p(x_1, f(x_4)) \doteq p(x_2, x_3)$$

zu lösen. Wir haben die folgenden Reduktions-Schritte:

$$\begin{array}{ll} & \langle \{p(x_1, f(x_4)) \doteq p(x_2, x_3)\}, [\,] \rangle \\ \rightsquigarrow & \langle \{x_1 \doteq x_2, f(x_4) \doteq x_3\}, [\,] \rangle \\ \rightsquigarrow & \langle \{f(x_4) \doteq x_3\}, [x_1 \mapsto x_2] \rangle \\ \rightsquigarrow & \langle \{x_3 \doteq f(x_4)\}, [x_1 \mapsto x_2] \rangle \\ \rightsquigarrow & \langle \{\}, [x_1 \mapsto x_2,\ x_3 \mapsto f(x_4)] \rangle \end{array}$$

In diesem Fall ist das Verfahren also erfolgreich und wir erhalten die Substitution

$$[x_1 \mapsto x_2,\ x_3 \mapsto f(x_4)]$$

als Lösung der oben gegebenen syntaktischen Gleichung. ◇

Beispiel: Wir geben ein weiteres Beispiel und betrachten das Gleichungs-System

$$E = \{p(h(x_1, c)) \doteq p(x_2),\ q(x_2, d) \doteq q(h(d, c), x_4)\}$$

Wir haben folgende Reduktions-Schritte:

$$\begin{aligned}
& \langle\{p(h(x_1,c)) \doteq p(x_2),\ q(x_2,d) \doteq q(h(d,c),x_4)\},[]\rangle \\
\leadsto\ & \langle\{p(h(x_1,c)) \doteq p(x_2),\ x_2 \doteq h(d,c),\ d \doteq x_4\},[]\rangle \\
\leadsto\ & \langle\{p(h(x_1,c)) \doteq p(x_2),\ x_2 \doteq h(d,c),\ x_4 \doteq d\},[]\rangle \\
\leadsto\ & \langle\{p(h(x_1,c)) \doteq p(x_2),\ x_2 \doteq h(d,c)\},[x_4 \mapsto d]\rangle \\
\leadsto\ & \langle\{p(h(x_1,c)) \doteq p(h(d,c))\},[x_4 \mapsto d,\ x_2 \mapsto h(d,c)]\rangle \\
\leadsto\ & \langle\{h(x_1,c) \doteq h(d,c)\},[x_4 \mapsto d,\ x_2 \mapsto h(d,c)]\rangle \\
\leadsto\ & \langle\{x_1 \doteq d,\ c \doteq c\},[x_4 \mapsto d,\ x_2 \mapsto h(d,c)]\rangle \\
\leadsto\ & \langle\{x_1 \doteq d,\},[x_4 \mapsto d,\ x_2 \mapsto h(d,c)]\rangle \\
\leadsto\ & \langle\{\},[x_4 \mapsto d,\ x_2 \mapsto h(d,c),\ x_1 \mapsto d]\rangle
\end{aligned}$$

Damit haben wir die Substitution $[x_4 \mapsto d,\ x_2 \mapsto h(d,c),\ x_1 \mapsto d]$ als Lösung des anfangs gegebenen syntaktischen Gleichungs-Systems gefunden. ◇

7.7 Ein Kalkül für die Prädikatenlogik ohne Gleichheit

In diesem Abschnitt setzen wir voraus, dass unsere Signatur Σ das Gleichheits-Zeichen nicht verwendet, denn durch diese Einschränkung wird es wesentlich einfacher, einen vollstängen Kalkül für die Prädikatenlogik einzuführen. Zwar gibt es auch für den Fall, dass die Signatur Σ das Gleichheits-Zeichen enthält, einen vollständigen Kalkül. Dieser ist allerdings deutlich aufwendiger als der Kalkül, den wir gleich einführen werden.

Definition 49 (Resolution) *Es gelte:*

1. k_1 *und* k_2 *sind prädikatenlogische Klauseln,*
2. $p(s_1, \cdots, s_n)$ *und* $p(t_1, \cdots, t_n)$ *sind atomare Formeln,*
3. *die syntaktische Gleichung* $p(s_1, \cdots, s_n) \doteq p(t_1, \cdots, t_n)$ *ist lösbar mit*

$$\mu = \texttt{mgu}\big(p(s_1, \cdots, s_n), p(t_1, \cdots, t_n)\big).$$

Dann ist

$$\frac{k_1 \cup \{p(s_1, \cdots, s_n)\} \qquad \{\neg p(t_1, \cdots, t_n)\} \cup k_2}{k_1\mu \cup k_2\mu}$$ *eine Anwendung der Resolutions-Regel.*

◇

Die Resolutions-Regel ist eine Kombination aus der Substitutions-Regel und der Schnitt-Regel. Die Substitutions-Regel hat die Form

$$\frac{k}{k\sigma}.$$

Hierbei ist k eine prädikatenlogische Klausel und σ ist eine Substitution. Unter Umständen kann es sein, dass wir bei der Anwendung der Resolutions-Regel die Variablen in einer der beiden Klauseln erst umbenennen müssen bevor wir die Regel anwenden können. Betrachten wir dazu ein Beispiel. Die Klausel-Menge

$$M = \Big\{ \{p(x)\}, \{\neg p(f(x))\} \Big\}$$

ist widersprüchlich. Wir können die Resolutions-Regel aber nicht unmittelbar anwenden, denn die syntaktische Gleichung

$$p(x) \doteq p(f(x))$$

ist unlösbar. Das liegt daran, dass **zufällig** in beiden Klauseln dieselbe Variable verwendet wird. Wenn wir die Variable x in der zweiten Klausel jedoch zu y umbenennen, erhalten wir die Klausel-Menge

$$\Big\{ \{p(x)\}, \{\neg p(f(y))\} \Big\}.$$

Hier können wir die Resolutions-Regel anwenden, denn die syntaktische Gleichung

$$p(x) \doteq p(f(y))$$

hat die Lösung $[x \mapsto f(y)]$. Dann erhalten wir

$$\{p(x)\}, \quad \{\neg p(f(y))\} \quad \vdash \quad \{\}.$$

und haben damit die Inkonsistenz der Klausel-Menge M nachgewiesen.

Die Resolutions-Regel alleine ist nicht ausreichend, um aus einer Klausel-Menge M, die inkonsistent ist, in jedem Fall die leere Klausel ableiten zu können: Wir brauchen noch eine zweite Regel. Um das einzusehen, betrachten wir die Klausel-Menge

$$M = \Big\{ \{p(f(x), y), p(u, g(v))\}, \{\neg p(f(x), y), \neg p(u, g(v))\} \Big\}$$

Wir werden gleich zeigen, dass die Menge M widersprüchlich ist. Man kann nachweisen, dass mit der Resolutions-Regel alleine ein solcher Nachweis nicht gelingt. Ein einfacher, aber für die Vorlesung zu aufwendiger Nachweis dieser Behauptung kann geführt werden, indem wir ausgehend von der Menge M alle möglichen Resolutions-Schritte durchführen. Dabei würden wir dann sehen, dass die leere Klausel nie berechnet werden kann. Wir stellen daher jetzt die Faktorisierungs-Regel vor, mir der wir später zeigen werden, dass M widersprüchlich ist.

Definition 50 (Faktorisierung) *Es gelte*

1. k *ist eine prädikatenlogische Klausel,*
2. $p(s_1, \cdots, s_n)$ *und* $p(t_1, \cdots, t_n)$ *sind atomare Formeln,*
3. *die syntaktische Gleichung* $p(s_1, \cdots, s_n) \doteq p(t_1, \cdots, t_n)$ *ist lösbar,*
4. $\mu = \texttt{mgu}\big(p(s_1, \cdots, s_n), p(t_1, \cdots, t_n)\big)$.

Dann sind

$$\frac{k \cup \{p(s_1, \cdots, s_n),\ p(t_1, \cdots, t_n)\}}{k\mu \cup \{p(s_1, \cdots, s_n)\mu\}} \quad \textit{und} \quad \frac{k \cup \{\neg p(s_1, \cdots, s_n),\ \neg p(t_1, \cdots, t_n)\}}{k\mu \cup \{\neg p(s_1, \cdots, s_n)\mu\}}$$

Anwendungen der Faktorisierungs-Regel. ◇

Wir zeigen, wie sich mit Resolutions- und Faktorisierungs-Regel die Widersprüchlichkeit der Menge M beweisen lässt.

1. Zunächst wenden wir die Faktorisierungs-Regel auf die erste Klausel an. Dazu berechnen wir den Unifikator

   $$\mu = \mathtt{mgu}\big(p(f(x), y), p(u, g(v))\big) = [y \mapsto g(v), u \mapsto f(x)].$$

   Damit können wir die Faktorisierungs-Regel anwenden:

   $$\big\{p(f(x), y), p(u, g(v))\big\} \quad \vdash \quad \big\{p(f(x), g(v))\big\}.$$

2. Jetzt wenden wir die Faktorisierungs-Regel auf die zweite Klausel an. Dazu berechnen wir den Unifikator

   $$\mu = \mathtt{mgu}\big(\neg p(f(x), y), \neg p(u, g(v))\big) = [y \mapsto g(v), u \mapsto f(x)].$$

   Damit können wir die Faktorisierungs-Regel anwenden:

   $$\big\{\neg p(f(x), y), \neg p(u, g(v))\big\} \quad \vdash \quad \big\{\neg p(f(x), g(v))\big\}.$$

3. Wir schließen den Beweis mit einer Anwendung der Resolutions-Regel ab. Der dabei verwendete Unifikator ist die leere Substitution, es gilt also $\mu = []$.

   $$\big\{p(f(x), g(v))\big\}, \quad \big\{\neg p(f(x), g(v))\big\} \quad \vdash \quad \{\}.$$

Ist M eine Menge von prädikatenlogischen Klauseln und ist k eine prädikatenlogische Klausel, die durch Anwendung der Resolutions-Regel und der Faktorisierungs-Regel aus M hergeleitet werden kann, so schreiben wir

$$M \vdash k.$$

Dies wird als M leitet k her gelesen.

Definition 51 (Allabschluss) *Ist k eine prädikatenlogische Klausel und ist $\{x_1, \cdots, x_n\}$ die Menge aller Variablen, die in k auftreten, so definieren wir den Allabschluss $\forall(k)$ der Klausel k als*

$$\forall(k) := \forall x_1 \colon \cdots \forall x_n \colon k.$$ ◇

Die für uns wesentlichen Eigenschaften des Beweis-Begriffs $M \vdash k$ werden in den folgenden beiden Sätzen zusammengefasst.

Satz 52 (Korrektheits-Satz)
Ist $M = \{k_1, \cdots, k_n\}$ eine Menge von Klauseln und gilt $M \vdash k$, so folgt

$$\models \forall(k_1) \wedge \cdots \wedge \forall(k_n) \rightarrow \forall(k).$$

Falls also eine Klausel k aus einer Menge M hergeleitet werden kann, so ist k tatsächlich eine Folgerung aus M. □

Die Umkehrung des obigen Korrektheits-Satzes gilt nur für die leere Klausel. Sie wurde 1965 von John A. Robinson bewiesen [Rob65].

Satz 53 (Widerlegungs-Vollständigkeit (Robinson, 1965))
Ist $M = \{k_1, \cdots, k_n\}$ eine Menge von Klauseln und gilt $\models \forall(k_1) \wedge \cdots \wedge \forall(k_n) \rightarrow \bot$, so folgt

$$M \vdash \{\}.$$ □

Damit haben wir nun ein Verfahren in der Hand, um für eine gegebene prädikatenlogischer Formel f die Frage, ob $\models f$ gilt, untersuchen zu können.

1. Wir berechnen zunächst die Skolem-Normalform von $\neg f$ und erhalten dabei so etwas wie

   $$\neg f \approx_e \forall x_1, \cdots, x_m \colon g.$$

2. Anschließend bringen wir die Matrix g in konjunktive Normalform:

   $$g \leftrightarrow k_1 \wedge \cdots \wedge k_n.$$

   Daher haben wir nun

   $$\neg f \approx_e k_1 \wedge \cdots \wedge k_n$$

   und es gilt:

   $$\models f \quad \text{g.d.w.} \quad \{\neg f\} \models \bot \quad \text{g.d.w.} \quad \{k_1, \cdots, k_n\} \models \bot.$$

3. Nach dem Korrektheits-Satz und dem Satz über die Widerlegungs-Vollständigkeit gilt

   $$\{k_1, \cdots, k_n\} \models \bot \quad \text{g.d.w.} \quad \{k_1, \cdots, k_n\} \vdash \bot.$$

   Wir versuchen also, nun die Widersprüchlichkeit der Menge $M = \{k_1, \cdots, k_n\}$ zu zeigen, indem wir aus M die leere Klausel ableiten. Wenn diese gelingt, haben wir damit die Allgemeingültigkeit der ursprünglich gegebenen Formel f gezeigt.

Beispiel: Zum Abschluss demonstrieren wir das skizzierte Verfahren an einem Beispiel. Wir gehen von folgenden Axiomen aus:

1. Jeder Drache ist glücklich, wenn alle seine Kinder fliegen können.
2. Rote Drachen können fliegen.
3. Die Kinder eines roten Drachens sind immer rot.

Wie werden zeigen, dass aus diesen Axiomen folgt, dass alle roten Drachen glücklich sind. Als erstes formalisieren wir die Axiome und die Behauptung in der Prädikatenlogik. Wir wählen die Signatur

$$\Sigma_{Drache} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$$

wobei die Mengen $\mathcal{V}$, $\mathcal{F}$, $\mathcal{P}$ und *arity* wie folgt definiert sind:

1. $\mathcal{V} := \{x, y, z\}$.
2. $\mathcal{F} = \{\}$.
3. $\mathcal{P} := \{\mathit{rot}, \mathit{fliegt}, \mathit{glücklich}, \mathit{kind}\}$.
4. $\mathit{arity} := \big\{\langle \mathit{rot}, 1\rangle, \langle \mathit{fliegt}, 1\rangle, \langle \mathit{glücklich}, 1\rangle, \langle \mathit{kind}, 2\rangle\big\}$

Das Prädikat $\mathit{kind}(x, y)$ soll genau dann wahr sein, wenn x ein Kind von y ist. Formalisieren wir die Axiome und die Behauptung, so erhalten wir die folgenden Formeln $f_1, \cdots, f_4$:

1. $f_1 := \forall x : \Big(\forall y : \big(\mathit{kind}(y, x) \to \mathit{fliegt}(y)\big) \to \mathit{glücklich}(x)\Big)$
2. $f_2 := \forall x : \big(\mathit{rot}(x) \to \mathit{fliegt}(x)\big)$
3. $f_3 := \forall x : \big(\mathit{rot}(x) \to \forall y : \big(\mathit{kind}(y, x) \to \mathit{rot}(y)\big)\big)$
4. $f_4 := \forall x : \big(\mathit{rot}(x) \to \mathit{glücklich}(x)\big)$

Wir wollen zeigen, dass die Formel

$$f := f_1 \wedge f_2 \wedge f_3 \to f_4$$

allgemeingültig ist. Wir betrachten also die Formel $\neg f$ und stellen fest

$$\neg f \leftrightarrow f_1 \wedge f_2 \wedge f_3 \wedge \neg f_4.$$

Als nächstes müssen wir diese Formel in eine Menge von Klauseln umformen. Da es sich hier um eine Konjunktion mehrerer Formeln handelt, können wir die einzelnen Formeln f_1, f_2, f_3 und $\neg f_4$ getrennt in Klauseln umwandeln.

1. Die Formel f_1 kann wie folgt umgeformt werden:

$$\begin{array}{rcl}
f_1 & = & \forall x : \Big(\forall y : \big(\mathit{kind}(y, x) \to \mathit{fliegt}(y)\big) \to \mathit{glücklich}(x)\Big) \\
& \leftrightarrow & \forall x : \Big(\neg\forall y : \big(\mathit{kind}(y, x) \to \mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
& \leftrightarrow & \forall x : \Big(\neg\forall y : \big(\neg\mathit{kind}(y, x) \vee \mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
& \leftrightarrow & \forall x : \Big(\exists y : \neg\big(\neg\mathit{kind}(y, x) \vee \mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
& \leftrightarrow & \forall x : \Big(\exists y : \big(\mathit{kind}(y, x) \wedge \neg\mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
& \leftrightarrow & \forall x : \exists y : \Big(\big(\mathit{kind}(y, x) \wedge \neg\mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
& \approx_e & \forall x : \Big(\big(\mathit{kind}(s(x), x) \wedge \neg\mathit{fliegt}(s(x))\big) \vee \mathit{glücklich}(x)\Big)
\end{array}$$

Im letzten Schritt haben wir dabei die Skolem-Funktion s mit $\mathit{arity}(s) = 1$ eingeführt. Anschaulich berechnet diese Funktion für jeden Drachen x, der nicht glücklich ist, ein Kind $s(x)$, das nicht fliegen kann. Wenn wir in der Matrix dieser Formel das "$\vee$" noch ausmultiplizieren, so erhalten wir die beiden Klauseln

$$k_1 := \{\mathit{kind}(s(x), x), \mathit{glücklich}(x)\},$$
$$k_2 := \{\neg\mathit{fliegt}(s(x)), \mathit{glücklich}(x)\}.$$

2. Analog finden wir für f_2:

$$\begin{array}{lcl} f_2 & = & \forall x : \big(\mathit{rot}(x) \rightarrow \mathit{fliegt}(x)\big) \\ & \leftrightarrow & \forall x : \big(\neg\mathit{rot}(x) \vee \mathit{fliegt}(x)\big) \end{array}$$

Damit ist f_2 zu folgender Klauseln äquivalent:

$$k_3 := \{\neg\mathit{rot}(x), \mathit{fliegt}(x)\}.$$

3. Für f_3 sehen wir:

$$\begin{array}{lcl} f_3 & = & \forall x : \Big(\mathit{rot}(x) \rightarrow \forall y : \big(\mathit{kind}(y, x) \rightarrow \mathit{rot}(y)\big)\Big) \\ & \leftrightarrow & \forall x : \Big(\neg\mathit{rot}(x) \vee \forall y : \big(\neg\mathit{kind}(y, x) \vee \mathit{rot}(y)\big)\Big) \\ & \leftrightarrow & \forall x : \forall y : \big(\neg\mathit{rot}(x) \vee \neg\mathit{kind}(y, x) \vee \mathit{rot}(y)\big) \end{array}$$

Das liefert die folgende Klausel:

$$k_4 := \{\neg\mathit{rot}(x), \neg\mathit{kind}(y, x), \mathit{rot}(y)\}.$$

4. Umformung der Negation von f_4 liefert:

$$\begin{array}{lcl} \neg f_4 & = & \neg\forall x : \big(\mathit{rot}(x) \rightarrow \mathit{glücklich}(x)\big) \\ & \leftrightarrow & \neg\forall x : \big(\neg\mathit{rot}(x) \vee \mathit{glücklich}(x)\big) \\ & \leftrightarrow & \exists x : \neg\big(\neg\mathit{rot}(x) \vee \mathit{glücklich}(x)\big) \\ & \leftrightarrow & \exists x : \big(\mathit{rot}(x) \wedge \neg\mathit{glücklich}(x)\big) \\ & \approx_e & \mathit{rot}(d) \wedge \neg\mathit{glücklich}(d) \end{array}$$

Die hier eingeführte Skolem-Konstante d steht für einen unglücklichen roten Drachen. Das führt zu den Klauseln

$$k_5 = \{\mathit{rot}(d)\},$$
$$k_6 = \{\neg\mathit{glücklich}(d)\}.$$

Wir müssen also untersuchen, ob die Menge M, die aus den folgenden Klauseln besteht, widersprüchlich ist:

1. $k_1 = \{\mathit{kind}(s(x), x),\ \mathit{glücklich}(x)\}$
2. $k_2 = \{\neg\mathit{fliegt}(s(x)),\ \mathit{glücklich}(x)\}$
3. $k_3 = \{\neg\mathit{rot}(x),\ \mathit{fliegt}(x)\}$
4. $k_4 = \{\neg\mathit{rot}(x),\ \neg\mathit{kind}(y, x),\ \mathit{rot}(y)\}$

5. $k_5 = \{rot(d)\}$
6. $k_6 = \{\neg glücklich(d)\}$

Sei also $M := \{k_1, k_2, k_3, k_4, k_5, k_6\}$. Wir zeigen, dass $M \vdash \bot$ gilt:

1. Es gilt
$$\mathtt{mgu}\big(rot(d), rot(x)\big) = [x \mapsto d].$$
Daher können wir die Resolutions-Regel auf die Klauseln k_5 und k_4 wie folgt anwenden:
$$\{rot(d)\},\ \{\neg rot(x), \neg kind(y,x), rot(y)\} \ \vdash\ \{\neg kind(y,d), rot(y)\}.$$
2. Wir wenden nun auf die resultierende Klausel und auf die Klausel k_1 die Resolutions-Regel an. Dazu berechnen wir zunächst
$$\mathtt{mgu}\big(kind(y,d), kind(s(x),x)\big) = [y \mapsto s(d), x \mapsto d].$$
Dann haben wir
$$\{\neg kind(y,d), rot(y)\},\ \{kind(s(x),x), glücklich(x)\} \ \vdash\ \{glücklich(d), rot(s(d))\}.$$
3. Jetzt wenden wir auf die eben abgeleitete Klausel und die Klausel k_6 die Resolutions-Regel an. Wir haben:
$$\mathtt{mgu}\big(glücklich(d), glücklich(d)\big) = []$$
Also erhalten wir
$$\{glücklich(d), rot(s(d))\},\ \{\neg glücklich(d)\} \ \vdash\ \{rot(s(d))\}.$$
4. Auf die Klausel $\{rot(s(d))\}$ und die Klausel k_3 wenden wir die Resolutions-Regel an. Zunächst haben wir
$$\mathtt{mgu}\big(rot(s(d)), \neg rot(x)\big) = [x \mapsto s(d)]$$
Also liefert die Anwendung der Resolutions-Regel:
$$\{rot(s(d))\},\ \{\neg rot(x), fliegt(x)\} \ \vdash\ \{fliegt(s(d))\}$$
5. Um die so erhaltenen Klausel $\{fliegt(s(d))\}$ mit der Klausel k_3 resolvieren zu können, berechnen wir
$$\mathtt{mgu}\big(fliegt(s(d)), fliegt(s(x))\big) = [x \mapsto d]$$
Dann liefert die Resolutions-Regel
$$\{fliegt(s(d))\},\ \{\neg fliegt(s(x)), glücklich(x)\} \ \vdash\ \{glücklich(d)\}.$$
6. Auf das Ergebnis $\{glücklich(d)\}$ und die Klausel k_6 können wir nun die Resolutions-Regel anwenden:
$$\{glücklich(d)\},\ \{\neg glücklich(d)\} \ \vdash\ \{\}.$$

Da wir im letzten Schritt die leere Klausel erhalten haben, ist insgesamt $M \vdash \bot$ nachgewiesen worden und damit haben wir gezeigt, dass alle kommunistischen Drachen glücklich sind. $\diamond$

Aufgabe 12: Die von Bertrant Russell definierte *Russell-Menge* R ist definiert als die Menge aller der Mengen, die sich nicht selbst enthalten. Damit gilt also

$$\forall x : \bigl(x \in R \leftrightarrow \neg x \in x\bigr).$$

Zeigen Sie mit Hilfe des in diesem Abschnitt definierten Kalküls, dass diese Formel widersprüchlich ist.

Aufgabe 13: Gegeben seien folgende Axiome:

1. Jeder Barbier rasiert alle Personen, die sich nicht selbst rasieren.
2. Kein Barbier rasiert jemanden, der sich selbst rasiert.

Zeigen Sie, dass aus diesen Axiomen logisch die folgende Aussage folgt:

> Alle Barbiere sind blond.

7.8 *Prover9* und *Mace4*

Der im letzten Abschnitt beschriebene Kalkül lässt sich automatisieren und bildet die Grundlage moderner automatischer Beweiser. Gleichzeitig lässt sich auch die Suche nach Gegenbeispielen automatisieren. Wir stellen in diesem Abschnitt zwei Systeme vor, die diesen Zwecken dienen.

1. *Prover9* dient dazu, automatisch prädikatenlogische Formeln zu beweisen.
2. *Mace4* untersucht, ob eine gegebene Menge prädikatenlogischer Formeln in einer endlichen Struktur erfüllbar ist. Gegebenenfalls wird diese Struktur berechnet.

Die beiden Programme *Prover9* und *Mace4* wurden von William McCune [McC10] entwickelt, stehen unter der GPL (*Gnu General Public Licence*) und können unter der Adresse

`http://www.cs.unm.edu/~mccune/prover9/download/`

im Quelltext heruntergeladen werden. Wir diskutieren zunächst *Prover9* und schauen uns anschließend *Mace4* an.

7.8.1 Der automatische Beweiser *Prover9*

Prover9 ist ein Programm, das als Eingabe zwei Mengen von Formeln bekommt. Die erste Menge von Formeln wird als Menge von *Axiomen* interpretiert, die zweite Menge von Formeln sind die zu beweisenden *Theoreme*, die aus den Axiomen gefolgert werden sollen. Wollen wir beispielsweise zeigen, dass in der Gruppen-Theorie aus der Existenz eines links-inversen Elements auch die Existenz eines rechts-inversen Elements folgt und dass außerdem das links-neutrale Element auch rechts-neutral ist, so können wir zunächst die Gruppen-Theorie wie folgt axiomatisieren:

1. $\forall x : e \cdot x = x$,
2. $\forall x : \exists y : y \cdot x = e$,
3. $\forall x : \forall y : \forall z : (x \cdot y) \cdot z = x \cdot (y \cdot z)$.

Wir müssen nun zeigen, dass aus diesen Axiomen die beiden Formeln

$$\forall x : x \cdot e = x \quad \text{und} \quad \forall x : \exists y : y \cdot x = e$$

logisch folgen. Wir können diese Formeln wie in Abbildung 7.19 auf Seite 173 gezeigt für *Prover9* darstellen. Der Anfang der Axiome wird in dieser Datei durch "`formulas(sos)`" eingeleitet und durch das Schlüsselwort "`end_of_list`" beendet. Zu beachten ist, dass sowohl die Schlüsselwörter als auch die einzelnen Formel jeweils durch einen Punkt "`.`" beendet werden. Die Axiome in den Zeilen 2, 3, und 4 drücken aus, dass

1. `e` ein links-neutrales Element ist,
2. zu jedem Element x ein links-inverses Element y existiert und
3. das Assoziativ-Gesetz gilt.

Aus diesen Axiomen folgt, dass das `e` auch ein rechts-neutrales Element ist und dass außerdem zu jedem Element x ein rechts-neutrales Element y existiert. Diese beiden Formeln sind die zu beweisenden Ziele und werden in der Datei durch "`formulas`(goal)" markiert. Trägt die in Abbildung 7.19 gezeigte Datei den Namen "`group2.in`", so können wir das Programm *Prover9* mit dem Befehl

```
prover9 -f group2.in
```

starten und erhalten als Ergebnis die Information, dass die beiden in Zeile 8 und 9 gezeigten Formeln tatsächlich aus den vorher angegebenen Axiomen folgen. Ist eine Formel nicht beweisbar, so gibt es zwei Möglichkeiten: In bestimmten Fällen kann *Prover9* tatsächlich erkennen, dass ein Beweis unmöglich ist. In diesem Fall bricht das Programm die Suche nach einem Beweis mit einer entsprechenden Meldung ab. Wenn die Dinge ungünstig liegen, ist es auf Grund der Unentscheidbarkeit der Prädikatenlogik nicht möglich zu erkennen, dass die Suche nach einem Beweis scheitern muss. In einem solchen Fall läuft das Programm solange weiter, bis kein freier Speicher mehr zur Verfügung steht und bricht dann mit einer Fehlermeldung ab.

```
1  formulas(sos).
2  all x (e * x = x).                             % left neutral
3  all x exists y (y * x = e).                    % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)). % associativity
5  end_of_list.
6
7  formulas(goals).
8  all x (x * e = x).                             % right neutral
9  all x exists y (x * y = e).                    % right inverse
10 end_of_list.
```

Figure 7.19: Textuelle Darstellung der Axiome der Gruppentheorie.

Prover9 versucht, einen indirekten Beweis zu führen. Zunächst werden die Axiome in prädikatenlogische Klauseln überführt. Dann wird jedes zu beweisenden Theorem negiert und die negierte Formel wird ebenfalls in Klauseln überführt. Anschließend versucht *Prover9* aus der Menge aller Axiome zusammen mit den Klauseln, die sich aus der Negation eines der zu beweisenden Theoreme ergeben, die leere Klausel herzuleiten. Gelingt dies, so ist bewiesen, dass das jeweilige Theorem tatsächlich aus den Axiomen folgt. Abbildung 7.20 zeigt eine Eingabe-Datei für *Prover9*, bei der versucht wird, das Kommutativ-Gesetz aus den Axiomen der Gruppentheorie zu folgern. Der Beweis-Versuch mit *Prover9* schlägt allerdings fehl. In diesem Fall wird die Beweissuche nicht endlos fortgesetzt. Dies liegt daran, dass es *Prover9* gelingt, in endlicher Zeit alle aus den gegebenen Voraussetzungen folgenden Formeln abzuleiten. Leider ist ein solcher Fall eher die Ausnahme als die Regel.

```
formulas(sos).
all x (e * x = x).                            % left neutral
all x exists y (y * x = e).                   % left inverse
all x all y all z ((x * y) * z = x * (y * z)). % associativity
end_of_list.

formulas(goals).
all x all y (x * y = y * x).                  % * is commutative
end_of_list.
```

Figure 7.20: Gilt das Kommutativ-Gesetz in allen Gruppen?

7.8.2 *Mace4*

Dauert ein Beweisversuch mit *Prover9* endlos, so ist zunächst nicht klar, ob das zu beweisende Theorem gilt. Um sicher zu sein, dass eine Formel nicht aus einer gegebenen Menge von Axiomen folgt, reicht es aus, eine Struktur zu konstruieren, in der alle Axiome erfüllt sind, in der das zu beweisende Theorem aber falsch ist. Das Programm *Mace4* dient genau dazu, solche Strukturen zu finden. Das funktioniert natürlich nur, solange die Strukturen endlich sind. Abbildung 7.21 zeigt eine Eingabe-Datei, mit deren Hilfe wir die Frage, ob es endliche nicht-kommutative Gruppen gibt, unter Verwendung von *Mace4* beantworten können. In den Zeilen 2, 3 und 4 stehen die Axiome der Gruppen-Theorie. Die Formel in Zeile 5 postuliert, dass für die beiden Elemente a und b das Kommutativ-Gesetz nicht gilt, dass also $a \cdot b \neq b \cdot a$ ist. Ist der in Abbildung 7.21 gezeigte Text in einer Datei mit dem Namen "*group.in*" gespeichert, so können wir *Mace4* durch das Kommando

mace4 -f group.in

starten. *Mace4* sucht für alle positiven natürlichen Zahlen $n = 1, 2, 3, \cdots$, ob es eine Struktur $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ mit $card(U) = n$ gibt, in der die angegebenen Formeln gelten. Bei $n = 6$ wird *Mace4* fündig und berechnet tatsächlich eine Gruppe mit 6 Elementen, in der das Kommutativ-Gesetz verletzt ist.

```
1 formulas(theory).
2 all x (e * x = x).                            % left neutral
3 all x exists y (y * x = e).                   % left inverse
4 all x all y all z ((x * y) * z = x * (y * z)). % associativity
5 a * b != b * a.                               % a and b do not commute
6 end_of_list.
```

Figure 7.21: Gibt es eine Gruppe, in der das Kommutativ-Gesetz nicht gilt?

Abbildung 7.22 zeigt einen Teil der von *Mace4* produzierten Ausgabe. Die Elemente der Gruppe sind die Zahlen $0, \cdots, 5$, die Konstante a ist das Element 0, b ist das Element 1, e ist das Element 2. Weiter sehen wir, dass das Inverse von 0 wieder 0 ist, das Inverse von 1 ist 1 das Inverse von 2 ist 2, das Inverse von 3 ist 4, das Inverse von 4 ist 3 und das Inverse von 5 ist 5. Die Multiplikation wird durch die folgende Gruppen-Tafel realisiert:

$\circ$	0	1	2	3	4	5
0	2	3	0	1	5	4
1	4	2	1	5	0	3
2	0	1	2	3	4	5
3	5	0	3	4	2	1
4	1	5	4	2	3	0
5	3	4	5	0	1	2

Diese Gruppen-Tafel zeigt, dass

$$a \circ b = 0 \circ 1 = 3, \quad \text{aber} \quad b \circ a = 1 \circ 0 = 4$$

gilt, mithin ist das Kommutativ-Gesetz tatsächlich verletzt.

```
============================== DOMAIN SIZE 6 =========================

=== Mace4 starting on domain size 6. ===

============================== MODEL =================================

interpretation( 6, [number=1, seconds=0], [

        function(a, [ 0 ]),

        function(b, [ 1 ]),

        function(e, [ 2 ]),

        function(f1(_), [ 0, 1, 2, 4, 3, 5 ]),

        function(*(_,_), [
                          2, 3, 0, 1, 5, 4,
                          4, 2, 1, 5, 0, 3,
                          0, 1, 2, 3, 4, 5,
                          5, 0, 3, 4, 2, 1,
                          1, 5, 4, 2, 3, 0,
                          3, 4, 5, 0, 1, 2 ])
]).

============================== end of model ==========================
```

Figure 7.22: Ausgabe von *Mace4*.

Bemerkung: Der Theorem-Beweiser *Prover9* ist ein Nachfolger des Theorem-Beweisers *Otter*. Mit Hilfe von *Otter* ist es William McCune 1996 gelungen, die Robbin'sche Vermutung zu beweisen [McC97]. Dieser Beweis war damals sogar der New York Times eine Schlagzeile wert, nachzulesen unter

`http://www.nytimes.com/library/cyber/week/1210math.html`.

Dies zeigt, dass automatische Theorem-Beweiser durchaus nützliche Werkzeuge sein können. Nichtsdestoweniger ist die Prädikatenlogik unentscheidbar und bisher sind nur wenige offene mathematische Probleme mit Hilfe von automatischen Beweisern gelöst worden. Das wird sich vermutlich auch in der näheren Zukunft nicht ändern. ◇

7.9 Reflexion

1. Was ist eine Signatur?
2. Wie haben wir die Menge $\mathcal{T}_\Sigma$ der Σ-Terme definiert?
3. Was ist eine atomare Formel?
4. Wie haben wir die Menge $\mathbb{F}_\Sigma$ der Σ-Formeln definiert?
5. Was ist eine Σ-Struktur?
6. Es sei $\mathcal{S}$ eine Σ-Struktur. Wie haben wir den Begriff der $\mathcal{S}$-Variablen-Belegung definiert?
7. Wie haben wir die Semantik von Σ-Formeln definiert?
8. Wann ist eine prädikatenlogische Formel allgemeingültig?
9. Was bedeutet die Schreibweise $\mathcal{S} \models F$ für eine Σ-Struktur $\mathcal{S}$ und eine Σ-Formel F?
10. Wann ist eine Menge von prädikatenlogischen Formeln unerfüllbar?
11. Was ist ein Constraint Satisfaction Problem?
12. Wie funktioniert Backtracking?
13. Warum kommt es beim Backtracking auf die Reihenfolge an, in der die verschiedenen Variablen instantiiert werden?
14. Was sind prädikatenlogische Klauseln und welche Schritte müssen wir durchführen, um eine gegebene prädikatenlogische Formel in eine erfüllbarkeits-äquivalente Menge von Klauseln zu überführen?
15. Was ist eine Substitution?
16. Was ist ein Unifikator?
17. Geben Sie die Regeln von Martelli und Montanari an!
18. Wie ist die Resolutions-Regel definiert und warum ist es eventuell erforderlich, Variablen umzubenennen, bevor die Resolutions-Regel angewendet werden kann?

19. Was ist die Faktorisierungs-Regel?

20. Wie gehen wir vor, wenn wir die Allgemeingültigkeit einer prädikatenlogischen Formel f nachweisen wollen?

Bibliography

[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.

[CPR11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.

[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.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[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.

[McC97] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19:263–276, December 1997.

[McC10] William McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[MGS96] Martin Müller, Thomas Glaß, and Karl Stroetmann. Automated modular termination proofs for real prolog programs. In Radhia Cousot and David A. Schmidt, editors, *SAS*, volume 1145 of *Lecture Notes in Computer Science*, pages 220–237. Springer, 1996.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[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.

[Rob65] John A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1996.

[Tur36] Alan M. Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.