Theoretical Computer Science I: Logic

— Winter 2018 —

DHBW Mannheim

Prof. Dr. Karl Stroetmann

October 10, 2018

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

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

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

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

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

```
git pull.
```

Contents

Chapter 1

Introduction

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

1.1 Motivation

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

Staggering Impact of IT Systems Gone Wrong

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

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

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

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

1.2 Overview

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

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

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

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

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

   "$\neg$", "$\wedge$", "$\vee$", "$\rightarrow$" 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 numbers via the property

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

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

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

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

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

and further define the set R as follows:

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

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

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

So we have shown that

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

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

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

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

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

defines a set. The expression

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

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

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

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

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

2.1 Defining Sets by Listing their Elements

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

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

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

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

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

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

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

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

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

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

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

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

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

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

since both sets contain the same elements.

2.2 Predefined Infinite Sets of Numbers

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

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

We write

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

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

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

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

2.3 The Axiom of Specification

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

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

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

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

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

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 "$x \in M \land p(x)$".

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

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

2.4 Power Sets

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

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

Example: We have

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

Furthermore, for any set M we have that

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

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

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

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

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

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.

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

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

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

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

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

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

then

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

◇

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

2.6 The Intersection of Sets

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

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

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

$$M \cap N = \{3,5\}.$$

◇

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

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

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

2.7 The Difference of Sets

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

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

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

$$M \backslash N = \{1,7\}.$$

◇

2.8 Image Sets

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

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

This set is also written as

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

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

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

Alternatively, we can define this set as

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

◇

2.9 Cartesian Products

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

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

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

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

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

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

To be more concise we usually write this as

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

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

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

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

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

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

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

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

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

2.10 Equality of Sets

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

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

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

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

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

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

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

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

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

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

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

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

2.11 Chapter Review

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

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

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

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 to experiment with set theory. This is the programming language Python, which has its own website at `python.org`. By programming in *Python*, students can get acquainted with set theory in a playful manner. Furthermore, as many interesting problems have a straightforward solution as *Python* programs, students can appreciate the usefulness of abstract notions from set theory by programming in *Python*. Furthermore, according to Philip Guo, 8 of the top 10 US universities teach *Python* in their introductory computer science courses.

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

3.1 Starting the Interpreter

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

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

The *Python* welcome message.

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

```
python
```

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

```
1 + 2
```

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

and press enter, we get the following output:

```
3
>>>
```

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

```
exit()
```

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

3.2 An Introduction to *Python*

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

3.2.1 Evaluating expressions

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

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

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

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

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

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

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

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

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

```
In [3]: import functools

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

```
Out[3]:
402387260077093773543702433923003985719374864210714632543799910429938512
398629020592044208486969404800479988610197196058631666872994808558901323
829669944590997424504087073759918823627727188732519779505950995276120874
975462497043601418278094646496291056393887437886487337119181045825783647
849977012476632889835955735432513185323958463075557409114262417474349347
553428646576611667797396668820291207379143853719588249808126867838374559
731746136085379534524221586593201928090878297308431392844403281231558611
036976801357304216168747609675871348312025478589320767169132448426236131
412508780208000261683151027341827977704784635868170164365024153691398281
264810213092761244896359928705114964975419909342221566832572080821333186
116811553615836546984046708975602900950537616475847728421889679646244945
160765353408198901385442487984959953319101723355556602139450399736280750
137837615307127761926849034352625200015888535147331611702103968175921510
907788019393178114194545257223865541461062892187960223838971476088506276
862967146674697562911234082439208160153780889893964518263243671616762179
168909779911903754031274622289988005195444414282012187361745992642956581
746628302955570299024324153181617210465832036786906117260158783520751516
284225540265170483304226143974286933061690897968482590125458327168226458
066526769958652682272807075781391858178889652208164348344825993266043367
660176999612831860788386150279465955131156552036093988180612138558600301
435694527224206344631797460594682573103790084024432438465657245014402821
885252470935190620929023136493273497565513958720559654228749774011413346
962715422845862377387538230483865688976461927383814900140767310446640259
899490222221765904339901886018566526485061799702356193897017860040811889
729918311021171229845901641921068884387121855646124960798722908519296819
372388642614839657382291123125024186649353143970137428531926649875337218
940694281434118520158014123344828015051399694290153483077644569099073152
433278288269864602789864321139083506217095002597389863554277196742822248
757586765752344220207573630569498825087968928162753848863396909959826280
956121450994871701244516461260379029309120889086942028510640182154399457
156805941872748998094254742173582401063677404595741785160829230135358081
840096996372524230560855903700624271243416909004153690105933983835777939
410970027753472000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
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 $\texttt{range}(a, b+1)$ 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 $\texttt{sum}(n)$ 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. $\texttt{sum}(n) = \texttt{sum}(n-1) + n \quad$ provided that $n > 0$.

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

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

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

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

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

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

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

3.2.2 Sets in *Python*

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

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

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

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

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

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

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

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

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

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

```
In [15]: A
```

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

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

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

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

Again A is not changed.

```
In [17]: A
```

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

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

```
In [18]: A - B
```

```
Out[18]: {1}
```

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

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

```
Out[19]: False
```

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

```
In [20]: 1 in A
```

```
Out[20]: True
```

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

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

```
Out[21]: True
```

3.2.3 Defining Sets via Selection and Images

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

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

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

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

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

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

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:

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

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

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

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

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

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

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

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

3.2.4 Computing the Power Set

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

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

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

If we combine these parts we get the following equation:

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

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

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

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

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

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

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

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

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

3.2.5 Pairs and Cartesian Products

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

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

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

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

```
In [33]: 1, 2
```

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

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

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

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

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

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

3.2.6 Tuples

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

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

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

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

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

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

Tuples can be concatenated using the operator "+":

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

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

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

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

```
Out[38]: 6
```

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

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

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

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

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

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

```
In [41]: T3
```

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

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

$L[a : b]$

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

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

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

Slicing works with negative indices, too:

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

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

3.2.7 Lists

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

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

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

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

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

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

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

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

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

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

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

```
Out[47]: 3
```

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

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

```
Out[48]: 3
```

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

```
Out[49]: 1
```

3.2.8 Boolean Operators

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

```
In [50]: True
```

```
Out[50]: True
```

```
In [51]: False
```

```
Out[51]: False
```

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

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

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

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

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

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

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

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

not True = False
not False = True
```

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

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

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

Out[55]: False

In [56]: 1 < 2

Out[56]: True

In [57]: 1 <= 2

Out[57]: True

In [58]: 1 > 2

Out[58]: False

In [59]: 1 >= 2

Out[59]: False
```

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

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

Out[60]: True
```

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

`all`(L)

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

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

Out[61]: True
```

3.2.9 Control Structures

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

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

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

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

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

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

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

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

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

1. We set n equal to 100 as we want to compute the set all prime numbers less or equal that 100.
2. `primes` is the list of numbers from 0 upto n, i.e. we have initially
$$\texttt{primes} = [0, 1, 2, \cdots, n]$$
Therefore, we have
$$\texttt{primes}[i] = i \quad \text{for all } i \in \{0, 1, \cdots, n\}.$$
The idea is to set `primes`$[i]$ to zero iff i is a proper product of two numbers.
3. To this end we iterate over all i and j from the set $\{2, \cdots, n\}$ and set the product `primes`$[i * j]$ to zero. This is achieved by the two `for` loops below.
4. Note that we have to check that the product $i * j$ is not bigger than n for otherwise we would get an out of range error when trying to assign `primes[i*j]`.
5. After the iteration, all non-prime elements greater than one of the list primes have been set to zero.
6. Finally, we compute the set of primes by collecting those elements that have not been set to 0.

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

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

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

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

2. When we iterate over j in the inner loop, it is sufficient if we start with $j = i$ since all products of the form $i * j$ where $j < i$ have already been eliminated at the time, when the multiples of i had been eliminated.

3. If `primes[`i`] = 0`, then i is not a prime and hence it has to be a product of two numbers a and b both of which are smaller than i. However, since all the multiples of a and b have already been eliminated, there is no point in eliminating the multiples of i since these are also mulples of both a and b and hence have already been eliminated. Therefore, if `primes[`i`] = 0` we can immediately jump to the next value of i. This is achieved by the `continue` statement in line 7 below.

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

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

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

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

3.2.10 Numerical Functions

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

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

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

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

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

The sine function is called as follows:

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

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

The cosine function is called as follows:

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

```
Out[70]: 1.0
```

The tangent function is called as follows:

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

```
Out[71]: 0.9999999999999999
```

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

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

```
Out[72]: 1.5707963267948966
```

```
In [73]: math.acos(1.0)
```

```
Out[73]: 0.0
```

```
In [74]: math.atan(1.0)
```

```
Out[74]: 0.7853981633974483
```

Euler's number e can be computed as follows:

```
In [75]: math.e
```

```
Out[75]: 2.718281828459045
```

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

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

```
Out[76]: 2.718281828459045
```

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

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

```
Out[77]: 2.0
```

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

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

```
Out[78]: 1.4142135623730951
```

3.2.11 Selection Sort

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

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

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

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 29 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='')
```

Implementing selection sort in *Python*.

3.3 Loading a Program

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

`import` *file*

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

```
import min_sort
```

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

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

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

3.4 Strings

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

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

returns the result

"abcuvw".

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

```
n * s;
```

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

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

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

```
"abc" * 3
```

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

s[k..l]

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

```
s = "abcdefgh"
```

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

"cde".

3.5 Computing with Unlimited Precision

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

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

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

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

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

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

```
5/6
```

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

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

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

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

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

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

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

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

Lets approximate e by the following sum:

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

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

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

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

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

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

Insert a "`.`" after the first digit:

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

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

```
2.718281828459045235360287471352662497757247093699959574966967627724076630353547
59457138217852516642742
```

3.6 Other References

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

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

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

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

2. *The Quick Python Book* written by Naomi R. Ceder [Ced18] is up to date and gives a concise introduction to *Python*. The book assumes that the reader has some prior programming experience. I would assume that most of our students have the necessary background to feel comfortable with this book.

3. *Learning Python* by Mark Lutz [Lut13] is aimed at the complete novice. It discusses everything in minute detail, albeit at the cost of 1648 pages.

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

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 34 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 34 shows the program `solve.py` that uses this approach to solve the equation $x = \cos(x)$.

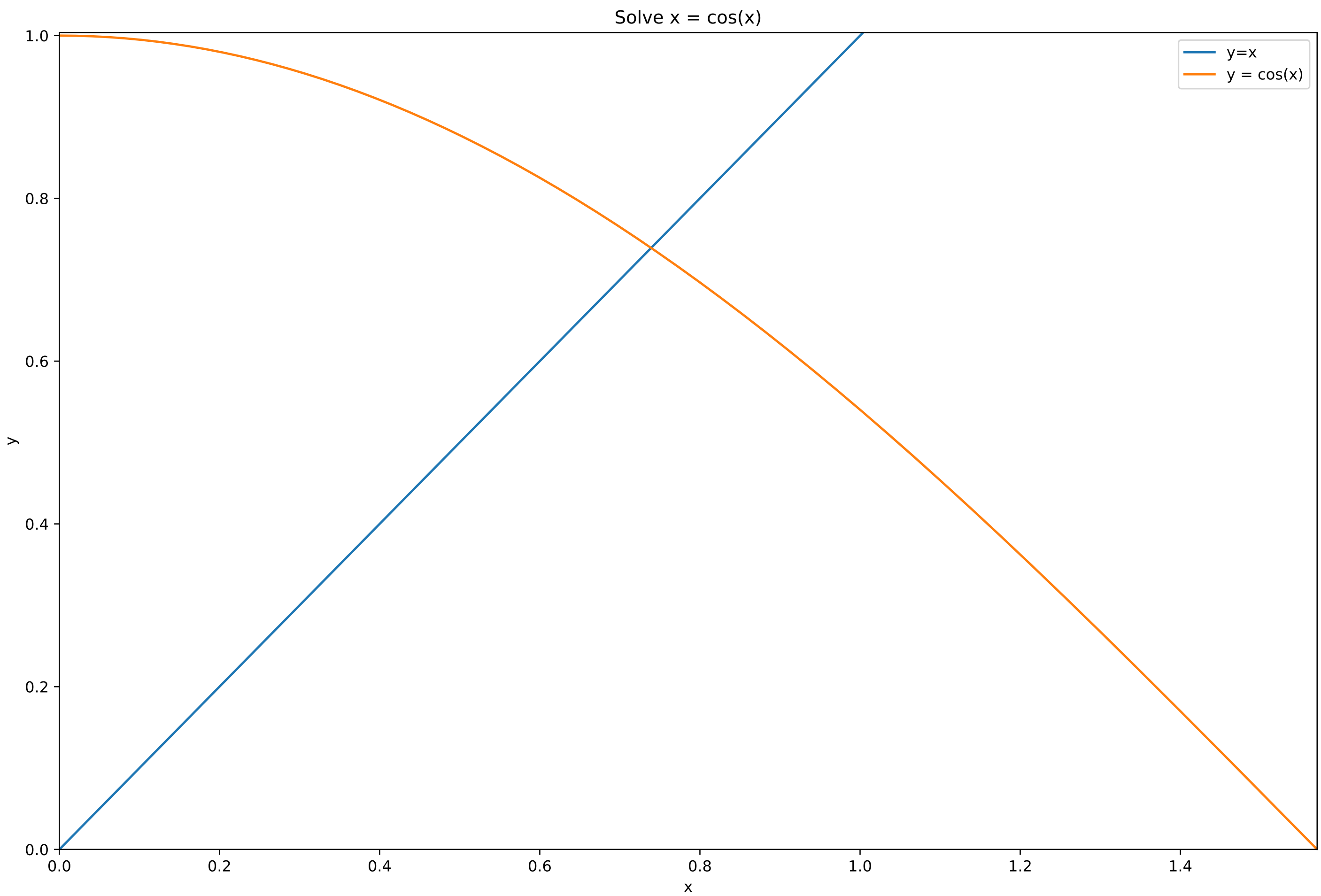
In this program, the iteration stops as soon as the difference between the variables `x` and `old_x` is less that $4 \cdot 10^{-16}$. Here, `x` corresponds to x_{n+1}, while `old_x` corresponds to x_n. Once the values of x_{n+1} and x_n are sufficiently close, the execution of the `while` loop terminates.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

Figure 4.3 on page 35 shows the program `fixpoint.py`. In this program we have implemented a function `solve` that takes two arguments.

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

$$f(x) = x.$$

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



The functions $y = x$ and $y = cos(x)$.

```
import math

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

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

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

A generic implementation of the fixed-point algorithm.

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

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

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

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

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

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

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

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

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

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

4.2 Case Study: Computation of Poker Probabilities

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

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

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

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

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

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

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))
```

Computing a probability in poker.

1. In line 1 the set `Values` is defined to be the set of all possible values that a card can take. In defining this set we have made use of the following abbreviations:

   (a) "T" is short for "Ten",

   (b) "J" is short for "Jack",

   (c) "Q" is short for "Queen",

   (d) "K" is short for "King", and

   (e) "A" is short for "Ace".

2. In line 2 the set `Suits` represents the possible suits of a card. Here, we have used the following abbreviations:

   (a) "c" is short for ♣, which is pronounced as club,

   (b) "h" is short for ♡, which is pronounced as heart,

   (c) "d" is short for ♢, which is pronounced as diamond, and

   (d) "s" is short for ♠, which is pronounced as spade.

3. Line 3 defines the set of all cards. This set is stored as the variable `Deck`. Every card is represented as a pair of the form $[v, s]$. Here, v is the value of the card, while s is its suit.

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

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

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

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

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

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

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

4.3 Finding a Path in a Graph

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

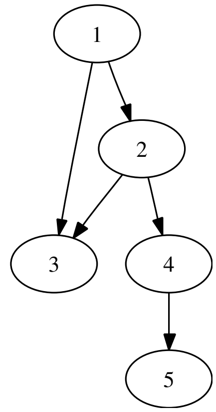
$$\mathsf{R} = \{\langle 1,2\rangle, \langle 2,3\rangle, \langle 1,3\rangle, \langle 2,4\rangle, \langle 4,5\rangle\}.$$

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 38. You should note that the connections between vertices that are given in this graph are unidirectional: While there is a connection from vertex 1 to vertex 2, there is no connection from vertex 2 to vertex 1.

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



A simple graph.

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

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

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

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

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

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

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

4.3.1 Computing the Transitive Closure of a Relation

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

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

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

Base Case: $n = 1$.

$R^1 := R$

Induction Step: $n \mapsto n+1$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In general, we can show by induction that

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

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

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

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

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

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

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

because we have:

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

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

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

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

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

But this implies that

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

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

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

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

The program `transitive-closure.py` that is shown in Figure 4.6 on page 42 shows an implementation of this idea. The program produces the following output:

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

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

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

4.3.2 Computing the Paths

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

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

```
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).union(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 );
```

Computing the transitive closure.

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

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

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

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:

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

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 43. 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 43 shows a cyclic graph. This graph is cyclic because it contains the path

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

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

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

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

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

1. In line 2 and 3, we compute only those paths that are not cyclic.

2. Line 6 defines a function `noCycle` that tests, whether the join `T1` $\oplus$ `T2` is cyclic. The join of `T1` and `T2` is cyclic iff the tuples `T1` and `T2` have more than one common element. The tuples `T1` and `T2` will always

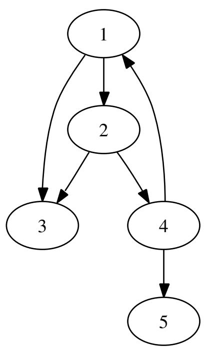
```
def findPaths(R):
    P = R;
    while True:
        oldP = P
        P    = R.union(pathProduct(P, R))
        if P == oldP:
            return P

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

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

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

Computing all connections.



A graph with a cycle.

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

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

Computing the connections in a cyclic graph.

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

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

Instead, we just want to compute the shortest path leading from `x` to `y`. Figure 4.10 on page 45 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 ⟨`start`⟩ that starts with vertex `start` and has length 0.
2. Line 5 tries to extend all previously computed paths by one step.
3. Line 6 selects all those paths from the set `P` that lead to the vertex `goal`. These paths are stored in the set `Found`.
4. Line 7 checks whether we have indeed found a path ending at `goal`. This is the case if the set `Found` is not empty. In this case, we return any of these paths.
5. If we have not yet found the vertex `goal` and, furthermore, we have not been able to find any new paths during this iteration, the procedure returns in line 10. As the `return` statement in line 11 does not return a value, the procedure will instead return the value `None`.

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

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

4.3.3 The Wolf, the Goat, and the Cabbage

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

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

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

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

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

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

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

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

Finding the shortest path between two vertices.

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

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

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

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

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

Therefore, we can implement the function problem as follows:

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

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

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

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

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

Let us explain this definition in detail:

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

   `'farmer' in B`.
5. Finally, the boat can only have two passengers. Therefore, we have added the condition

   `len(B) <= 2`.

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

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

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

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

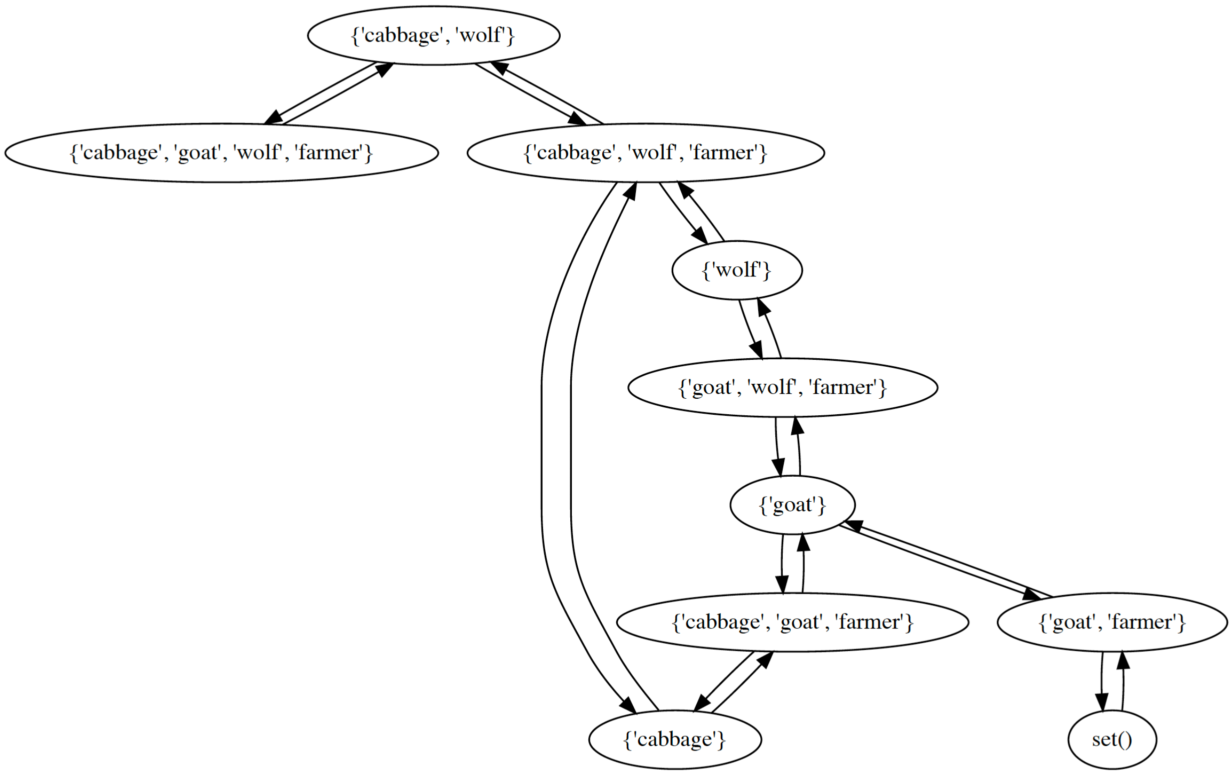
Finally, the start state has all objects on the left side. Therefore, we have

```
start = All
```

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

```
goal = {}
```

Figure 4.12 on page 47 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.



The relation R shown as a directed graph.

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

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

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"}
```

A schedule for the agricultural economist.

4.4 Symbolic Differentiation

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

`"x * exp(x)"`,

interprets this string as describing the real valued function

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

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

1. Every number $c \in \mathbb{R}$ is an arithmetic expression.
2. Every variable v is an arithmetic expression.
3. If s and t are arithmetic expressions, then

   $s + t, \quad s - t, \quad s * t, \quad s/t, \quad \text{and} \quad s ** t$

   are arithmetic expressions. Here $s ** t$ is interpreted as s^t.
4. If e is an arithmetic expression, then both

   $\exp(e) \quad \text{and} \quad \ln(e)$

   are arithmetic expressions.

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

1. The first argument `expr` represents 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 a nested tuple.

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

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

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.

We close this section by showing an example that demonstrates the power of matching. The function `diff` that is shown in Figure 4.14 on page 51 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
$$\ln\bigl(f(x)\bigr)$$
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
$$\frac{\mathrm{d}}{\mathrm{d}x}\ln\bigl(f(x)\bigr) = \frac{\frac{\mathrm{d}}{\mathrm{d}x}f(x)}{f(x)}$$

7. Line 26 - 29 deals with the case where `e` has the form $\exp\bigl(f(x)\bigr)$. 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
$$\frac{\mathrm{d}}{\mathrm{d}x}\exp\bigl(f(x)\bigr) = \left(\frac{\mathrm{d}}{\mathrm{d}x}f(x)\right)\cdot\exp\bigl(f(x)\bigr)$$

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

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

```python
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
```

A function for symbolic differentiation

```python
import exprParser as ep

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

Testing symbolic differentiation.

4.5 Reflection

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

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

Chapter 5

Grenzen der Berechenbarkeit

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

5.1 Das Halte-Problem

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

5.1.1 Informale Betrachtungen zum Halte-Problem

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

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

Nehmen wir nun an, wir hätten ein schlaues Programm, nennen wir es `stops`, das als Eingabe eine *Python* Funktion f und ein Argument a verarbeitet und das uns die Frage, ob die Berechnung von $f(a)$ terminiert,

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

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

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

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

Eine Funktion zur Überprüfung der von Legendre aufgestellten Vermutung.

beantworten kann. Die Idee wäre, dass die Funktion `stops` die folgende Spezifikation erfüllt:

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

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

`stops(findCounterExample, 1)`

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

$$\texttt{stops(findCounterExample, 1)} = \texttt{true}$$

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

`stops(findCounterExample, 1)`

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

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

5.1.2 Formale Analyse des Halte-Problems

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

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

```
procedure(x) { ··· }
```

hat und sich als SETLX-Funktion parsen lässt. Die Menge der Test-Funktionen bezeichnen wir mit *TF*. $\diamond$

Beispiele:

1. $s_1 :=$ "`procedure(x) { return 0; }`"

   s_1 ist eine (sehr einfache) Test-Funktion.

2. $s_2 :=$ "`procedure(x) { while (true) { x := x + 1; } }`"

   s_2 ist ebenfalls eine Test-Funktion. Offenbar liefert diese Test-Funktion nie ein Ergebnis, aber für die Frage, ob s_2 eine Test-Funktion ist oder nicht, ist dies irrelevant.

3. $s_3 :=$ "`procedure(x) { return ++x; }`"

   s_3 ist keine Test-Funktion, denn da SETLX den Präfix-Operator "++" nicht unterstützt, lässt sich der String s_3 nicht fehlerfrei parsen.

4. $s_4 :=$ "`procedure(x, y) { return x + y; }`"

   s_4 ist keine Test-Funktion, denn ein String ist nur dann eine Test-Funktion, wenn die durch den String definierte Funktion mit genau einen Parameter aufgerufen wird.

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

Notation 2 ($\leadsto$, $\downarrow$, $\uparrow$) Ist f eine SETLX-Funktion, die k Argumente verarbeitet und sind $a_1, \cdots, a_k$ mögliche Argumente, mit denen wir diese Funktion aufrufen können, so schreiben wir

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

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

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

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

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

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

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

1. `procedure(x) { return 0; }(1)` $\leadsto 0$
2. `procedure(x) { return 0; }(42)` $\downarrow$
3. `procedure(x) { while (true) { x := x + 1; } }(0)` $\uparrow$
4. `procedure(x) { while (true) { x := x + 1; } }(true)` $\uparrow$

   Im letzten Fall führt die Zuweisung "`x := x + 1;`" zu einem Fehler, denn es ist in SETLX nicht möglich, eine Zahl zu einem Booleschen Wert zu addieren.

Das Halte-Problem für SETLX-Funktionen ist die Frage, ob es eine `SetlX`-Funktion

$\texttt{stops := procedure}(t,\ a)\ \{\ \cdots\ \}$

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

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

   Der Aufruf `stops(`t`, `a`)` liefert genau dann den Wert 2 zurück, wenn t keine Test-Funktion ist oder der Aufruf $t(a)$ zu einem Fehler führt.

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

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

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

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

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

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

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

```
turing := "procedure(x) {
               result := stops(x, x);
               if (result == 1) {
                   while (true) {
                       print("... looping ...");
                   }
               }
               return result;
           }";
```

Die Definition des Strings `turing`.

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

$\texttt{turing} \in \mathit{TF}.$

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

`stops(turing, turing);`

Da `turing` eine Test-Funktion ist und der Aufruf von `turing` mit dem Argument `turing` auch nicht zu einem Fehler führen darf, können nur zwei Fälle auftreten:

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

Diese beiden Fälle analysieren wir nun im Detail:

1. `stops(turing, turing)` $\leadsto 0$.

   Nach der Spezifikation von `stops` bedeutet dies

   `turing(turing)` $\uparrow$

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

   Damit ist der erste Fall ausgeschlossen.

2. `stops(turing, turing)` $\leadsto 1$.

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

   `turing(turing)` $\downarrow$

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

   Damit ist der zweite Fall ausgeschlossen.

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

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

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

5.2 Unlösbarkeit des Äquivalenz-Problems

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

Definition 4 ($\simeq$) Es seien t_1 und t_2 zwei SETLX-Funktionen und $a_1, \cdots, a_k$ seien Argumente, mit denen wir diese Funktionen füttern können. Wir definieren

$$t_1(a_1, \cdots, a_k) \simeq t_2(a_1, \cdots, a_k)$$

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

1. $t_1(a_1, \cdots, a_k) \uparrow \quad \wedge \quad t_2(a_1, \cdots, a_k) \uparrow$,
   beide Funktionen divergieren also für die gegebenen Argumente.

2. $\exists r : \Big(t_1(a_1, \cdots, a_k) \leadsto r \ \wedge \ t_2(a_1, \cdots, a_k) \leadsto r \Big)$,
   die Funktionen liefern also für die gegebenen Argumente das gleiche Ergebnis.

In diesem Fall sagen wir, dass die beiden Funktions-Aufrufe $t_1(a_1, \cdots, a_k) \simeq t_2(a_1, \cdots, a_k)$ partiell äquivalent sind. □

Wir kommen jetzt zum Äquivalenz-Problem. Die Funktion `equal`, die die Form

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

hat, möge folgender Spezifikation genügen:

1. $p_1 \notin TF \ \vee \ p_2 \notin TF \quad \Leftrightarrow \quad \texttt{equal}(p_1, p_2, a) \leadsto 2$.
2. Falls
   (a) $p_1 \in TF$,
   (b) $p_2 \in TF$ und
   (c) $p_1(a) \simeq p_2(a)$

   gilt, dann muss gelten:
   $$\texttt{equal}(p_1, p_2, a) \leadsto 1.$$
3. Ansonsten gilt
   $$\texttt{equal}(p_1, p_2, a) \leadsto 0.$$

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

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

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

```
1  stops := procedure(p, a) {
2      f := "procedure(x) { while (true) { x := 1; } }";
3      e := equal(f, p, a);
4      if (e == 2) {
5          return 2;
6      } else {
7          return 1 - e;
8      }
9  };
```

Eine Implementierung der Funktion `stops`.

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

```
procedure(x) { while (true) { x := 1; } };
```

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

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

5.3 Reflexion

Beantworten Sie die folgenden Fragen.

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

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.

[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.

[Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 83, 1953.

[Tur36] Alan M. Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.